

Programmer en Assembleur

illustré avec le jeu d'instruction du Z-80

Alain Pinaud

Editions du P.S.I.

Programmer
en ASSEMBLEUR

Programmer en APL — Daniel-Jean David
Programmer en Assembleur — Alain Pinaud
Programmer en BASIC — Michel Plouin
Le BASIC et ses fichiers - Tome 1 : Méthodes pratiques — Jacques Boisgontier
Le BASIC et ses fichiers - Tome 2 : Programmes — Jacques Boisgontier
Programmer en FORTRAN — Daniel-Jean David
Programmer en L.S.E. — Stéphane Berche et Yves Noyelle
Programmer en PASCAL — Daniel-Jean David et Jean-Luc Deschamps
Comment programmer — Jean-Claude Barbance
La découverte de l'Apple soft — Frédéric Lévy et Dominique Schraen
La pratique de l'Apple II - volume I — Nicole Bréaud-Pouliquen
La pratique de l'Apple II - volume II — Nicole Bréaud-Pouliquen
La pratique de l'Apple II - volume III — Nicole Bréaud-Pouliquen
La pratique du LX500 - volume I — Alain Séméteys et Francis Vasse
La pratique du Sharp MZ-80K - volume I — Jean-Pierre Lhoir
La découverte du P.C. 1211 — Jean-Pierre Richard
La découverte du P.E.T. — Daniel-Jean David
La pratique du P.E.T./C.B.M. - volume I — Daniel-Jean David
La pratique du P.E.T./C.B.M. - volume II — Daniel-Jean David
La découverte de la TI-57 — Xavier de la Tullaye
La pratique du TRS-80 - volume I — Pierre Giraud et Alain Pinaud
La pratique du TRS-80 - volume II — Pierre Giraud et Alain Pinaud
La pratique du TRS-80 - volume III — Pierre Giraud et Alain Pinaud
Comprendre les microprocesseurs — Roland Dubois

Collection guides pratiques

La réalisation des programmes — Michel Benelfoul
Méthodes de calcul numérique — Claude Nowakowski

Collection programmes

Mathématiques et statistiques — Hervé Haut
Jeux, trucs et comptes (P.E.T./C.B.M.) — Michel Benelfoul

VISA POUR L'INFORMATIQUE — Jean-Michel Jego
MON ORDINATEUR — Jean-Claude Barbance

ISBN : 2-86595-014-X

Imprimé en France

Programmer en ASSEMBLEUR

par

Alain Pinaud

illustré avec le jeu d'instruction du Z-80



Editions du P.S.I.

1982

Alain Pinaud, *35 ans, est informaticien chez le constructeur Français d'ordinateurs CII-HONEYWELL BULL, où il participe à l'étude et à la réalisation de systèmes spéciaux, tant sur le plan matériel que logiciel.*

S O M M A I R E

AVANT-PROPOS

Ce livre s'adresse à la "cohorte grossissante des malheureux" qui, bien que possédant les éléments d'un langage évolué d'ordinateur (Basic, par exemple), hésitent encore à s'engager dans la droite et juste voie de l'assembleur...

Vous me direz qu'ils ont raison d'hésiter, après tout ce qu'on raconte :

- que l'assembleur n'est pas à la portée de l'amateur et est affaire de professionnels,
- que l'assembleur est réservé à l'élite,
- que l'assembleur est terriblement complexe et nécessite de très longues études,
- et blablabla, et blablabla...

Un vieux proverbe chinois dit : "*Ce qu'un sot peut faire, un autre le peut aussi*", et lorsqu'on voit le nombre de sots qui savent programmer en assembleur, il y a lieu de s'étonner que tant de personnes intelligentes ne le sachent pas encore !

Parmi ces sots - les mêmes que l'on rencontre aussi dans certaines sphères des mathématiques - il y a ceux qui redressent derrière eux le buisson d'épines qu'ils ont franchi, comme pour dérouter ceux qui suivent, et qui s'exclament par moments : "*Oui... vous pouvez me suivre... mais la route est longue et difficile... trèèès difficile...*".

Enfin, au nombre des sots, figurent aussi quelques abrutis, particulièrement dégénérés, dans la liste desquels s'inscrit l'auteur, qui osent encore penser que l'assembleur est à la portée de tous, pour peu que l'on consacre le temps nécessaire à son étude, et en l'abordant de manière résolue et sans complexes.

Que cet ouvrage modeste et ridicule soit l'impulsion, la "claque dans le dos" qui vous propulse en direction de la voie royale..., la droite et juste voie de l'assembleur.

C'est un "sot" souhait... mais c'est un "saut" qu'il faut faire !

INTRODUCTION

Un édifice, si impressionnant soit-il, n'est formé que de l'assemblage d'un ensemble de pierres ou de briques, et comme il faut bien partir de quelque chose, il sera supposé ici que ***le lecteur est familiarisé avec un langage évolué très simple, comme le Basic*** qui est le plus répandu actuellement et est implanté sur la plupart des ordinateurs individuels du commerce.

Bien entendu, cette connaissance d'un langage évolué n'est pas une condition exclusive pour aborder l'assembleur, mais elle permettra, lorsque le besoin s'en fera sentir, de développer certaines analogies entre ces deux formes de langage, propres à éclaircir ou à consolider le cours de notre étude.

Le lecteur non familiarisé avec des notions telles que binaire, bit, base, hexadécimal etc... a, à sa disposition en fin de volume, des annexes destinées à lui en faciliter la compréhension.

L'objectif principal de cet ouvrage étant de donner les rudiments permettant de programmer en assembleur dans le sens général des termes, il fallait, dans les exemples concrets, prendre pour support les instructions d'un assembleur existant (et il y en a des tas...).

Une autre démarche aurait été de créer, pour les besoins de la cause, des instructions hypothétiques (en Français, par exemple). Mais cela aurait été reculer pour mieux sauter.

Mieux vaut s'habituer, le plus tôt possible, à la terminologie existante et, après l'avoir assimilée, il sera possible de rêver un peu...

Si l'on veut faire l'effort de comprendre un aborigène d'Australie, on ne va pas commencer par exiger de lui qu'il comprenne le Français, n'est-ce pas ? Le microprocesseur est né en parlant Américain, il faut se faire à cette idée.

Quel assembleur existant fallait-il donc choisir ? Celui du ***microprocesseur Z80 de Zilog*** a été retenu pour deux raisons primordiales ; Il figure parmi les plus répandus, et son jeu d'instructions est le plus large dans sa catégorie, et inclut

celui de son cousin le 8080 d'Intel, également très répandu, et du plus récent 8085 à l'exception toutefois de deux instructions (mais pas trop de technique pour l'instant...!).

En cherchant bien, il y aurait même une troisième raison, mais nous n'en parlerons pas...!*

Enfin, il faut dire une chose : lorsque l'on connaît les principes de base du langage assembleur, ces mêmes principes restent valables sur tel ou tel ordinateur, qu'il s'agisse d'un 8 bits ou d'un 64 bits.

Les seules différences résideront dans la puissance, la souplesse et l'étendue du jeu d'instructions, ainsi, bien sûr, que dans la rapidité d'exécution de ces instructions.

Mais si la multiplication ou la division ne figurent pas au nombre des instructions d'un assembleur 8 bits, certaines instructions propres aux "8 bits" (échange, registres, manipulations de bits, pile... tututut... doucement la technique !), ne sont pas toujours présentes chez les "gros". Et c'est parfois assez gênant.

Aussi, il n'y a pas à avoir de complexes outre mesure !

Le temps est maintenant venu de passer aux choses sérieuses :

j'assemble,

tu assembles,

il assemble...

* *Note de l'Editeur : "Non ! l'auteur n'a pas d'actions Zilog. Il possède un TRS-80 !".*

CHAPITRE 1

DEFINITIONS ET RAPPELS DE BASE

Il est encore un peu tôt pour donner une définition de l'assembleur. Aussi, en attendant, nous examinerons les cas dans lesquels son emploi se justifie. Mais essayons tout de même de faire une petite approche.

IL ETAIT UNE FOIS...

Monsieur Yves Venelse avait une sainte horreur des robinets qui fuient. Rien de plus agaçant que ces POKE... POKE... De quoi devenir fou furieux ! Son cri de colère fit entrer LET, la femme de chambre (en fait, elle s'appelait Colette), qui s'enquit aussitôt :

- *"Monsieur m'a demandé ?"*
- *"REM... oui"*

  dit-il en s'éclaircissant la voix. Puis il reprit :

- *"Je veux que ce robinet soit réparé avant ce soir. Compris, LET ?".*

Elle marmonna un *"Bien, Monsieur"* et disparut.

Le soit, en rentrant, Yves Venelse constata distraitement que le robinet ne fuyait plus, et n'y prêta plus attention, ignorant pourtant quel travail cela avait été : il avait fallu rechercher un plombier libre, disposé à faire tant de chemin pour un simple robinet, en présence duquel il devait déclarer : *"que c'était un vieux modèle", "qu'il n'avait pas de quoi réparer", "qu'il fallait qu'il retourne à la ville"* et encore *"que cela allait coûter très cher"*... sans compter les inondations causées par ce maladroit... Ah ! Monsieur a le beau rôle !

J.P. Hachel, lui aussi, avait horreur des robinets qui fuient. Rien de plus agaçant que ces POP... POP...

Bricoleur et méticuleux de nature, il préférait faire ce travail lui-même. Après avoir coupé l'eau, il s'empara de sa boîte à outils, y préleva la clé appropriée et dévissa la tête du robinet. Il retira le joint coupable et le remplaça par un joint neuf, qu'il avait gardé bien précieusement dans une petite boîte. Il remonta la tête du robinet, la bloqua à l'aide de la clé, et restitua le circuit d'alimentation d'eau. Le malheur était réparé !

Voila pour la petite histoire, mais vous n'en aurez pas toujours des comme-çà...! A présent, si vous ne savez toujours pas ce qu'est l'assembleur, du moins savez-vous réparer un robinet qui fuit !

Au travers de ces deux histoires, nous pouvons faire les constatations suivantes

- *Yves Venelse* parle un langage évolué. L'ordre "*Je veux que ce robinet...*" est très clair et suffisant pour que l'action qui en résulte soit correctement interprétée. Monsieur Venelse n'a que faire, par contre, des petits à-côtés concernant le plombier, et encore moins le joint du robinet. A-t'il besoin d'ailleurs de savoir ce qu'est un joint ? Entre le moment où l'ordre est donné et celui où le robinet est réparé, il s'écoule un certain temps, et il s'exécute un certain nombre d'actions que Monsieur Venelse ignore - et veut ignorer. La seule chose qui compte pour lui, c'est le résultat.

- *J.P. Hachel*, lui, attache autant d'importance à ce qu'il fait qu'au résultat final. Il ne parle même pas, ou s'il parle c'est intérieurement. Il doit se dire des choses comme :

  - couper l'eau,
  - prendre la clé de 15,
  - non ce n'est pas la bonne...
  - essayer la 16... OK.
  - dévisser la tête... tourner... encore...
  - retirer le joint,
  - etc...

Son langage n'est pas très évolué, mais chaque "phrase" donne lieu à une petite action précise, sans équivoque, nous dirons une "micro-action". Encore que "couper l'eau" soit en fait un ensemble de micro-actions pouvant s'exprimer ainsi :

- tourner le robinet vers la droite et continuer jusqu'à ce qu'il soit bloqué...

Plus tard, nous appellerons cela **un sous-programme**.

Voyons maintenant ce que nous pouvons tirer de cette histoire :

- Si Y. Venelse parle Basic ou quelque chose d'approchant, J.P. Hachel parle Assembleur.

- Les actions déclenchées par l'ordre de Y. Venelse seront, en fin de compte, réalisées en "Assembleur".

- Le travail de Y. Venelse se borne à peu de choses. Il suffit qu'il dise "*Je veux que...*". La réalisation, par contre, est plus laborieuse. Mais qu'importe.

- Le travail de J.P. Hachel, lui, est beaucoup plus important, mais en revanche la réalisation est très rapide.

- On retrouve un peu aussi la différence d'esprit pouvant exister entre les deux langages : l'un clair et net, facile à formuler, l'autre laborieux, méticuleux, une petite bombe que l'on prépare avec soin...

Dans quels cas se justifie donc l'emploi de l'Assembleur ?

- Chaque fois que le facteur ***temps*** occupe un rôle primordial. Entre le temps d'exécution d'un programme fonctionnant sous interpréteur Basic et celui d'un programme directement écrit en assembleur, il peut y avoir un rapport énorme (de une à plusieurs dizaines).

- Chaque fois que le facteur ***espace*** occupe un rôle primordial. La taille d'un programme Basic (source) ajoutée à celle de son interpréteur, sera toujours supérieure à celle du même programme (accomplissant les même choses), réalisé en assembleur.

- Chaque fois qu'il est nécessaire de réaliser certaines ***fonctionnalités inconnues*** de l'interpréteur, ou impossibles à réaliser (pour les raisons précédentes, par exemple) avec un langage évolué.

Bien entendu, il est toujours possible - et cela se fait fréquemment - de concilier les deux formes de langage (évolué et assembleur), en écrivant l'ossature ou squelette du programme en Basic par exemple, ce dernier faisant appel, lorsque cela s'avère nécessaire, à de petits programmes réalisés en assembleur. Nous examinerons ces possibilités plus attentivement dans un lointain chapitre.

UN PETIT COMPRIME AVEC UN PEU D'EAU SUCREE

Il semble nécessaire de faire un peu d'anatomie, maintenant. Grossièrement, l'ordinateur est compose d'un ***microprocesseur*** qui est en quelque sorte son cerveau, et d'un bloc ***mémoire*** de taille variable, dans lequel sont stockés programmes et informations.

Afin de dialoguer avec l'extérieur (l'homme, par exemple), il doit être doté d'organes appelés ***périphériques***, qui peuvent se limiter à un simple clavier (entrée dialogue), et à un écran vidéo (sortie dialogue).

Accessoirement, sa mémoire peut se prolonger au moyen d'unités de cassettes ou de disquettes, qui ont également pour rôle de conserver - mais de manière plus durable - programmes et informations.

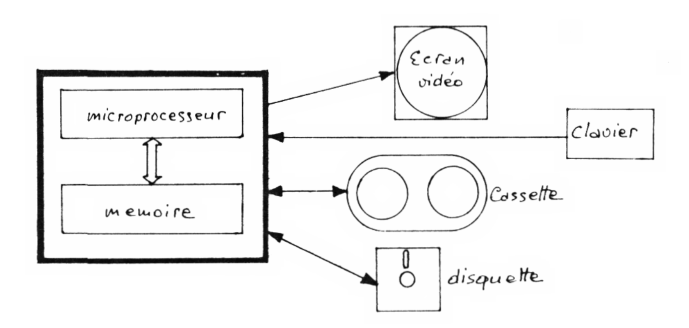


FIGURE 1

Dès que l'ordinateur est mis sous tension, le microprocesseur cherche à exécuter (c'est un besoin physique) le ***programme*** se trouvant au début de la mémoire (adresse 0).

Ce programme, afin de pouvoir être traité et exécuté par le microprocesseur, doit obligatoirement être composé d'***instructions*** connues du microprocesseur, celles pour l'exécution desquelles il a été conçu et qui lui sont propres (chaque type de microprocesseur a les siennes).

Et tout comme l'ADN détermine notre personnalité, ces instructions détermineront la "personnalité" du microprocesseur, ses performances, sa puissance, son "intelligence".

Chacune des instructions produit, au sein de l'ordinateur, une "micro-action" bien précise, et c'est la somme de ces micro-actions qui produira une action perceptible par l'homme, par exemple l'affichage d'un message sur l'écran, ou la saisie d'un message au clavier, ou bien encore la lecture ou l'écriture d'une information sur une unité de cassette.

Toutefois, l'action résultante peut ne pas être directement perçue. C'est le cas, par exemple, d'un calcul complexe ou d'un tri de nombres en mémoire.

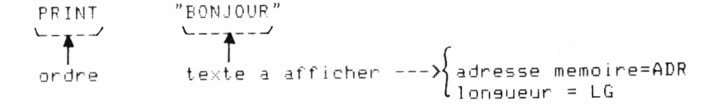
Lorsque l'homme recherche un nom dans sa mémoire, ou bien qu'il l'écrit sur une feuille de papier, il s'exécute, au niveau cérébral, des milliers de micro-actions. Dans ce dernier exemple, on peut penser que le cerveau sélectionne un "programme d'écriture", lequel, en liaison perpétuelle avec la mémoire, lance une foule d'ordres ou d'instructions qui vont provoquer les micro-actions nécessaires à la commande de certains muscles spécialisés, afin de coordonner les mouvements du poignet, de la main et des doigts, cependant que l'oeil, enregistrant les moindres déplacements, les transmet au cerveau afin de corriger une éventuelle "erreur de direction", ou de moduler la pression de la plume sur le papier.

La prochaine fois que vous frapperez : PRINT "BONJOUR" sur le clavier de votre ordinateur, vous ne verrez peut-être plus les choses de la même façon...

En effet, pour l'ordinateur, ce simple ordre Basic va provoquer tout un remue-ménage intérieur, un peu analogue à celui qui doit exister inconsciemment dans votre cerveau lorsque quelqu'un vous dit : "*ECRIS... ceci ou cela...*".

Essayons un peu de comprendre ce qui se passe réellement au niveau de la machine.

La phrase : PRINT "BONJOUR" entrée au clavier est transférée dans la mémoire de l'ordinateur par le programme interpréteur Basic. Après analyse de l'ordre (PRINT), le contrôle est donné à un petit programme spécialisé (sous-programme), dont le rôle est d'afficher des caractères sur l'écran (Nous l'appellerons "AFFICH"). Pour ce faire, il est nécessaire de communiquer à ce sous-programme l'adresse mémoire (car dans ce cas, nous travaillons en relation directe avec la mémoire) contenant le texte à afficher ainsi que sa longueur.



Le sous-programme exécute alors les opérations suivantes :

1. Prendre le caractère se trouvant à l'adresse mémoire ADR,
2. Appeler un autre sous-programme pour afficher ce caractère sur l'écran (appelons-le "ECRAN"),
3. Ajouter 1 à la valeur d'adresse ADR,
4. Soustraire 1 à la longueur LG,
5. Si LG est différent de zéro, retourner au point 1, sinon retourner au programme ayant provoqué l'appel.

Ces opérations peuvent être représentées graphiquement au moyen d'un ***organigramme***. C'est plus facile à comprendre, il suffit de suivre les flèches...

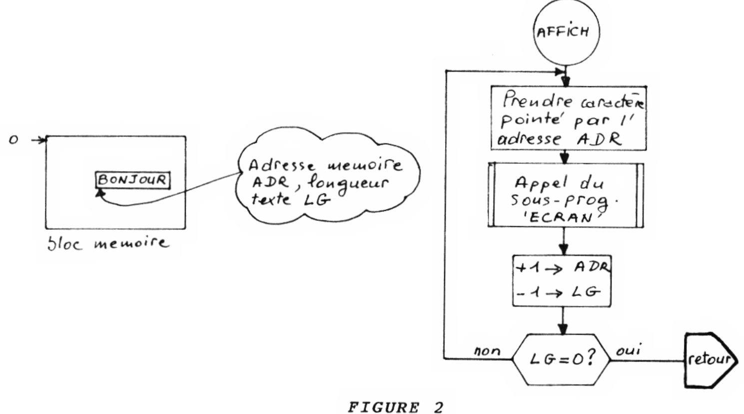


FIGURE 2

A titre documentaire, voici le programme Basic qui permettrait de se rapprocher au maximum de cet organigramme :

```
10 A$ = "BONJOUR"          ' DEFINITION DU TEXTE
20 ADR = 1                 ' POINTEUR ADRESSE
30 LG = 7                  ' LONGUEUR (OU LG = LEN(A$) )
40 GOSUB 100               ' APPEL S/P AFFICH
50 ....
      .
      .
      .
100 ' SOUS PROGRAMME AFFICH
102 ' DONNEES D'ENTREE: ADR = POINTEUR ADRESSE
103 '                   LG  = LONGUEUR TEXTE
104 '
110 C$ = MID$(A$,ADR,1)    ' PRENDRE CARACTERE POINTE PAR ADR
120 PRINT C$;              ' AFFICHAGE ECRAN
130 ADR = ADR + 1          ' POINTEUR ADRESSE + 1
140 LG = LG - 1            ' LONGUEUR - 1
150 IF LG <> 0 THEN 110    ' TEST SI FIN TEXTE
160 RETURN                 ' RETOUR AU PROGRAMME PRINCIPAL
```

C'est évidemment utiliser le Basic au minimum de ses possibilités, puisqu'il suffit d'écrire PRINT "BONJOUR" pour avoir le même résultat... mais ce programme nous a permis de "descendre un barreau" de l'échelle de l'évolution des langages de l'informatique. Et ce n'est pas fini ! Arrêtons-nous un peu pour respirer.

Nous disions précédemment qu'il fallait communiquer au sous-programme AFFICH les données concernant l'adresse et la longueur du texte à afficher.

Dans le programme Basic ci-dessus, nous avons appelé ces données ADR et LG. Mais si cela est suffisamment clair en Basic, cela ne l'est pas du tout au niveau de la machine.

L'ordinateur, lui, ne connaît que les adresses mémoire et les ***registres***.

Il faudra donc spécifier au départ :

- l'adresse du texte à afficher se trouve à l'emplacement mémoire 12693, par exemple, et la longueur de ce texte à l'emplacement mémoire 15312.

Il est plus facile de dire :

- l'adresse du texte à afficher se trouve dans le registre X et la longueur dans le registre A.

Ce principe de "boîte à lettres" doit évidemment être connu du programme et du sous-programmes appelés. Mais que sont donc les registres ?

Ce sont des sortes de mémoires rapides utilisées par le microprocesseur pour conserver temporairement les informations et qu'il utilise un peu comme bloc-notes. Mais au lieu d'y accéder par une adresse comme pour la mémoire, on y accède par

leurs noms : A, B, H, L par exemple, désignent quatre registres du microprocesseur Z-80.

Le nombre de ces registres, leurs tailles et leurs appellations varient d'un microprocesseur à l'autre. Contrairement au bloc mémoire qui est séparé, les registres sont à l'intérieur même du microprocesseur. Le jeu d'instructions utilise, bien entendu, ces différents registres.

Mais revenons à notre organigramme de la figure 2. Comment s'exprimer pour "dire" au microprocesseur : "*prendre caractère pointé par l'adresse ADR*" ?

Nous pouvons le faire au moyen d'une instruction créée spécialement pour cela, et connue du microprocesseur :

LD A,(HL)

et qui signifie : charger (LOAD en Anglais → LD) le registre A avec le contenu ( ) de l'adresse mémoire qui se trouve dans le double registre HL.

Là, nous venons brusquement de "descendre l'échelle" mais nous parlons assembleur ! Celui du Z-80.

Dans ce cas précis, le registre HL joue le rôle de "boîte à lettres" évoqué plus haut, et contient l'adresse ADR pointant sur le texte "BONJOUR".

Bien entendu, il est à la charge du programme appelant de charger ce registre antérieurement à l'appel du sous-programme. Notons au passage que cette forme de langage, si elle devient compréhensible à l'ordinateur, l'est de moins en moins pour l'homme...

Après exécution de cette instruction, le registre A contiendra le premier caractère du texte, c'est-à-dire la lettre 'B'.

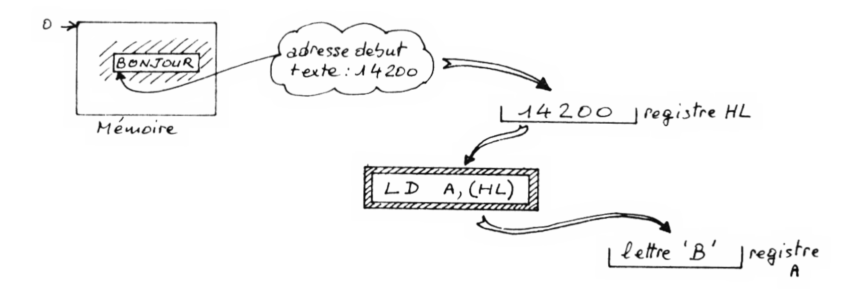


FIGURE 3

"Oui, mais...", direz-vous justement, *"je croyais que le microprocesseur ne "comprenait" réellement que les symboles 0 et 1 du système binaire ? Dans ce cas, comment peut-il comprendre quelque chose comme : LD A,(HL)...?"*

En effet, il ne peut pas le comprendre, et c'est bien là le drame...!

En réalité, le code de l'instruction LD A,(HL) équivaut, (pour le Z-80), à la codification binaire :

0 1 1 1 1 1 1 0

c'est-à-dire 7E en système hexadécimal.

C'est lorsque le microprocesseur trouve cette configuration binaire en mémoire, il exécute ce que nous, humains, nous appelons l'instruction LD A,(HL).

Comme il serait fastidieux de programmer un ordinateur en binaire, les concepteurs de microprocesseurs ont imaginé et écrit un programme qui réalise cette conversion : On dit à ce programme "LD A,(HL)" et il traduit "01111110"...

Ce programme, c'est l'assembleur

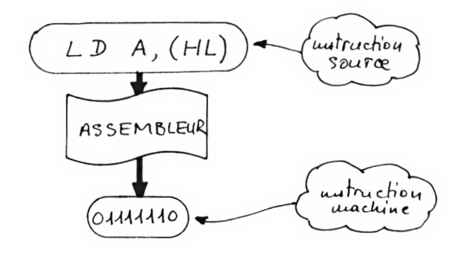


FIGURE 4

Chaque constructeur de microprocesseur doit également fournir le programme assembleur correspondant. En effet, sans ce dernier, le microprocesseur serait inexploitable.

Mais ce programme assembleur, dans quel langage est-il écrit? (m'énerve celui-là avec ses questions). En assembleur peut-être ? Exact. Mais il aurait très bien pu être écrit en Basic ou en Fortran... Oui, mais l'interpréteur Basic, dans quel langage est-il écrit, lui ? Hum... en assembleur.

En fait, le ***premier*** assembleur du ***premier*** microprocesseur a dû être écrit sur un autre ordinateur (qui n'avait pas de microprocesseur, bien sûr !)... et le ***premier*** assembleur du ***premier*** ordinateur, lui, a dû être écrit directement en binaire... Il n'y avait pas d'autres solutions !
Les instructions fournies à l'assembleur sont appelées instructions ***source***, et celles produites par l'assembleur, instructions ***machine***.

L'ensemble des instructions 'source' forme le ***code source*** et l'ensemble des instructions 'machine' forme le ***code machine*** ou ***code objet***. On rencontre aussi les appellations 'code externe' pour le code source, et 'code interne' pour le code objet.

Pouvons-nous donner une définition de l'assembleur, maintenant ?

Oui. Nous dirons que l'assembleur est un ***programme*** qui traduit les instructions écrites en langage source en leur équivalent en langage binaire (en établissant une correspondance biunivoque entre les instructions source et les instructions machine). Ce qui signifie qu'à chaque instruction source (entrée), il produira une instruction machine (sortie) directement interprétable par l'ordinateur pour lequel il est conçu.

L'ENVIRONNEMENT PROGRAMME

Contrairement aux langages évolués, l'utilisation de l'assembleur appelle des notions -nous dirons semi-techniques - qu'il faut éclaircir dès à présent et qui sont liées de très près à la machine.

La mémoire est, vous le savez probablement, le support qui permet de conserver et de retenir dans le temps l'information. Au cours de notre étude, nous utiliserons surtout la mémoire dite "centrale" qui est directement liée au microprocesseur, et qui contient programmes et données.

On accède à cette mémoire au moyen d'une adresse prenant les valeurs 0 (début mémoire) à FFFF (fin mémoire), soit 65536 "cases", ce qui est, en général, la valeur limite pour la plupart des microprocesseurs courants.

Comme la programmation en assembleur s'éloigne de l'homme pour se rapprocher de la machine, il est plus pratique d'utiliser le système de numération hexadécimal pour exprimer une adresse ou une donnée mémoire. Prenons-en donc l'habitude ! (Voir annexe I).

Le bloc de mémoire centrale peut être comparé à un meuble à tiroirs, chaque tiroir représentant une case mémoire appelée ***octet***.

Chaque octet à son tour contient huit informations binaires appelées ***bits***.

Chaque bit, que l'on peut considérer comme "l'atome de l'information", est indissecable et peut prendre l'une des deux valeurs binaires 0 ou 1.

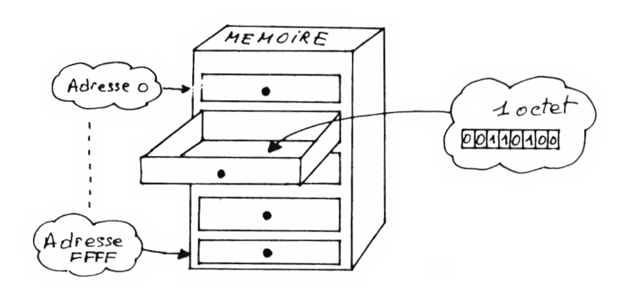


FIGURE 5

Dans un exemple précédent, le texte "BONJOUR" était rangé à partir de l'adresse mémoire 14200 (pardon, 3778H !). Cela signifie que le caractère "B" était à l'adresse 3778H, le caractère "O" à l'adresse 3779H, le caractère "N" à l'adresse 377AH... ETC. Nous verrons plus loin que l'on ne peut ranger qu'un seul symbole de l'alphabet dans un octet.

Le texte "BONJOUR" représente une information de type ***donnée*** par opposition à une information de type ***instruction***, ces deux types pouvant coexister dans la mémoire. Mais attention ! Le microprocesseur ne peut exécuter que des instructions !.

Lorsque, par erreur, il est amené à exécuter des données (ce qui peut arriver hélas...), la suite des évènements n'est pas triste du tout... et conduit ordinairement à ce que les programmeurs nomment le "plantage"...

Vous savez.. : on part à la pêche, tranquillement. On jette la ligne. Cinq minutes : rien. Dix minutes : rien. Une heure : rien. Alors on tire la ligne : plus de ver, plus d'hameçon, plus de bouchon... D'ailleurs il n'y a pas de rivière, pas plus que de ligne... Le pêcheur ? Quel pêcheur ? Y-a pas d'pêcheur. Rien que les petits oiseaux... Cui...cui...cui...

Souvenons-nous donc que les données sont destinées à être exploitées par les instructions et non pas substituées à elles, à moins bien sûr que ce ne soit volontaire. J'ai même vu des programmeurs vicieux...

Les ***registres*** peuvent aussi être assimilés à des cases mémoires contenant un ou deux octets. Ils sont utilisés par le microprocesseur et par les instructions du programme machine comme mémoires temporaires.

Ainsi, lorsque l'on fait une multiplication sur le papier, il est nécessaire d'écrire les résultats partiels afin de les additionner. Mais une fois le résultat obtenu, ils ne sont plus nécessaires et peuvent être effacés et oubliés.

En général, l'accès aux registres se fait, non par une adresse comme pour la mémoire, mais par un nom symbolique : A, X, HL,SP...

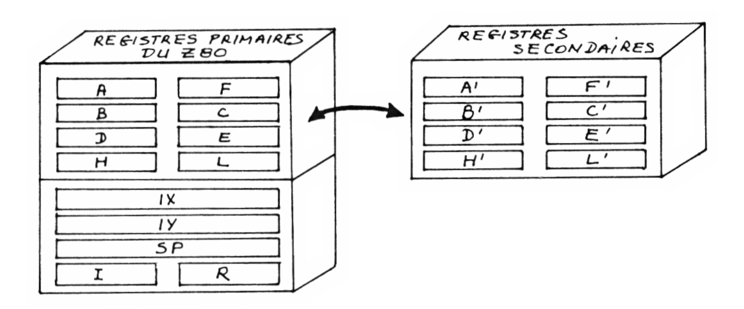


FIGURE 6

Le microprocesseur Z80 possède une seconde "banque" de registres nommés A' à L', qui peut se substituer, par une simple instruction machine, aux huit registres A à L. Après exécution de cette instruction, les registres A à L deviennent A' à L' et inversement.

Comme l'indique la figure 6, certains registres sont simples (8 bits) et d'autres doubles (16 bits).

De plus, certaines instructions considèrent les registres A et F, B et C, D et E, H et L, comme des registres doubles : AF, BC, DE et HL.

Par exemple, l'instruction :

```
LD    B,4
```

charge la valeur 4 dans le registre B, alors que l'instruction

```
LD    BC,1234H
```

charge la valeur 1234 H dans le double registre BC, ce qui reviendrait à faire :

```
LD    B,12H
LD    C,34H
```

en utilisant les instructions simple registre.

Quittons provisoirement les registres pour parler de ce que l'on nomme ***la pile*** (stack, en Anglais) et que l'on ne doit pas confondre avec celle de un volt cinq...

L'une des difficultés de la programmation en assembleur est la gestion de l'espace mémoire. Pour ranger une information, il faut savoir à quelle adresse mémoire la ranger afin de pouvoir la retrouver plus tard.

Avec la pile, c'est plus simple. Comme son nom l'indique, on "empile" les informations à ranger, les unes après les autres, à la manière d'une pile d'assiettes, sans avoir à se soucier de l'adresse mémoire à laquelle elles sont réellement,

et pour reprendre ces informations, on "dépile".

Exemple :

Ranger les doubles registres BC et DE, exécuter ensuite un traitement donné, puis recharger les registres BC et DE avec leurs valeurs initiales.

Il suffit de faire :

```
PUSH BC      ; rangement en pile de BC
PUSH DE      ; rangement en pile de DE
   :         ; traitement
POP  DE      ; rappel de DE
POP  BC      ; rappel de BC
```

Vous avez bien compris : l'instruction ***PUSH*** "empile" et l'instruction ***POP*** "dépile".

Mais attention à l'ordre : si l'on pose d'abord l'"assiette" BC et que l'on pose par dessus l'"assiette" DE, il faudra retirer DE afin de pouvoir reprendre BC... C'est logique !

Dans le petit exemple précédent, il faut supposer que le traitement séparant l'empilage du dépilage ne modifie pas la pile, bien entendu.

Mais cette pile, où se trouve-t'elle ? En mémoire. Une instruction spéciale permet d'en fixer l'origine. L'un des registres du microprocesseur, appelé SP (stack pointer = pointeur de pile) doit être chargé convenablement et antérieurement à toute utilisation des instructions faisant appel à la pile.

```
LD    SP,4000H     ; la pile sera à l'adresse 4000H
LD    BC,1234H     ; on charge 1234H dans BC
LD    DE,5678H     ; on charge 5678H dans DE
PUSH  BC           ; mise en pile de BC
PUSH  DE           ; mise en pile de DE
```

Après exécution de ces cinq instructions, la mémoire contient les informations suivantes :

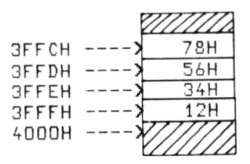


FIGURE 7

Nous constatons que les informations y sont rangées de bas en haut (la pile d'assiettes...). A chaque rangement dans la pile, le microprocesseur retire 1 au contenu du registre SP, afin de tenir un pointeur à jour. Après exécution des instructions ci-dessus, le registre SP contiendra la valeur 3FFCH. Il suffira d'exécuter deux instructions POP pour ramener SP à sa valeur initiale (4000H).

Il est à noter que, dans ce cas, les informations rangées aux adresses 3FFCH à 3FFFH ne seront pas altérées par ces deux instructions, mais qu'elles seront "écrasées" par les empilages suivants.

Comme il n'est pas toujours facile d'en déterminer la taille a priori, la pile est en général placée en "fond de mémoire", c'est-à-dire aux adresses fortes, afin de ne gêner personne.

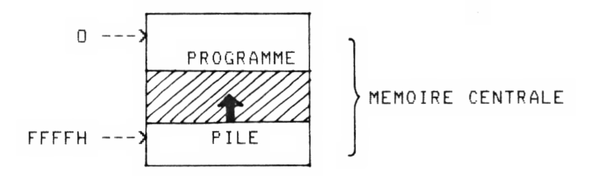
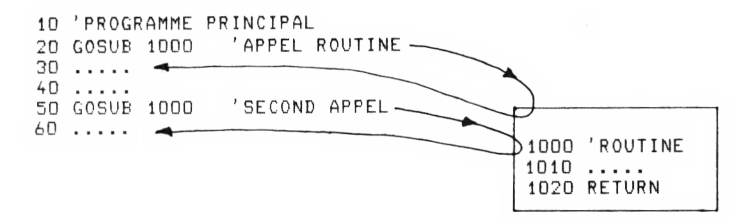


FIGURE 8

Nous allons voir maintenant que la pile est utilisée - de manière discrète, il est vrai - par les ***sous-programmes***.

Vous connaissez certainement l'instruction Basic "GOSUB", qui permet d'interrompre provisoirement le cours d'un programme pour aller exécuter une "routine" ou "sous-programme" - sorte de programme utilitaire, dont on a besoin très souvent -, et revenir au point interrompu du programme principal ?

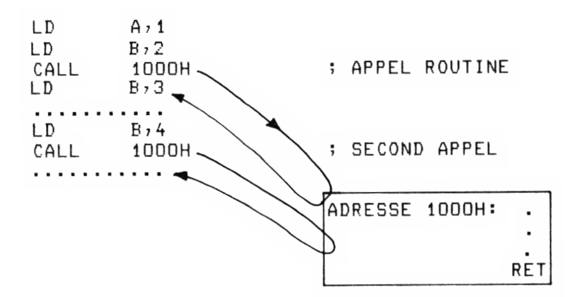


Cet artifice évite, en effet, l'écriture de séquences répétitives, et améliore la compréhension et la lisibilité des programmes qui deviennent ainsi plus structurés.

Il existe une fonction équivalente en assembleur :

- l'instruction ***CALL*** *permet d'appeler la routine,*

- l'instruction ***RET*** *provoque le retour au point d'appel*



Lorsque le microprocesseur rencontre l'instruction CALL, il place automatiquement dans la pile (par un PUSH invisible, en quelque sorte) l'adresse mémoire qui pointe sur l'instruction suivant le CALL (LD B,3 dans notre exemple), et part exécuter la routine située à l'adresse 1000H (par exemple).

A la fin de cette routine, l'instruction RET provoque un dépilage (POP invisible), et le programme poursuit son exécution par l'instruction située derrière le CALL.

Comme dans le cas des instructions PUSH et POP, il est nécessaire que le registre SP ait été initialisé précédemment à tout appel de sous-programme.

Côté routine, il faudra évidemment surveiller les mouvements de pile, sous peine de ne pas retourner à l'endroit voulu ! Il n'y est pas interdit d'exécuter des PUSH, mais à condition de faire le nombre de POP voulu avant d'atteindre l'instruction RET.

Revenons un peu en arrière maintenant, là où nous avions rangé le texte "BONJOUR".

Nous avions vu que les lettres composant ce texte étaient rangées consécutivement en mémoire. Mais sous quelle forme ?

Tout est question de convention en informatique : le code de l'instruction Z80 "LD A,(HL)" est 7EH, eh bien celui de la lettre "A" sera 41H, mais cette fois... il s'agit d'un code international, d'une importance primordiale dans tout ce qui touche les ordinateurs et la transmission de données : ***Le code ASCII***.

A.S.C.I.I. est formé des initiales des mots ***A****merican* ***S****tandard* ***C****ode for* ***I****nformation* ***I****nterchange* ce qui, traduit du Peau-Rouge, signifie sensiblement : Code Standard Américain pour l'Echange d'Information (les signaux de fumée étaient-ils en ASCII ?). L'intérêt, c'est qu'il est connu et interprété par pratiquement tous les organes périphériques aux ordinateurs.

Si l'on appuie sur la touche "A" d'un clavier relié à un ordinateur, il en sortira... devinez quoi ? ... Le code 41H.

De même, si l'on envoie ce code à un terminal vidéo (ou à une télétype), il affichera (ou imprimera) la lettre "A". C'est tout de même beau d'être compris dans ce monde d'électrons...!

Le code ASCII est formé de 128 combinaisons binaires permettant de décrire (voir en annexe III) :

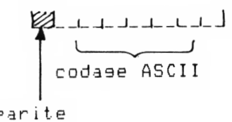
- les dix chiffres Arabes (0 à 9),
- des symboles spéciaux = , $, %, &, ...),
- des codes fonctionnels (retour ligne, saut de page sur une imprimante, etc...).

Hélas, cent fois hélas !... Cet "Espéranto" du traitement de l'information est parfois violé par certains constructeurs de périphériques qui trouvent plus logique d'imprimer un "[" à la place d'un "↑", ou quelqu'autre fantaisie de ce style. ∂'#h&< soient-ils... !

Heureusement, le microprocesseur ne comprend pas l'ASCII, ainsi il ne risque pas d'y avoir de problème, et il est à la charge du programme d'interpréter le codage des caractères.

Il existe d'autres formes de codages, mais l'ASCII étant le plus répandu dans le domaine des "micros", nous ne les aborderons pas ici.

En ASCII, chaque symbole, graphisme ou fonction, est codé sur un octet dont il utilise les 7 derniers bits. Pour cette raison, on l'appelle parfois : Code à 7 éléments. Ceci est suffisant pour coder les 128 combinaisons possibles, et permet de réserver le huitième bit pour une fonction de contrôle de parité.



Lorsque l'on doit transmettre des informations codées en ASCII à un organe périphérique éloigné (par exemple situé au bout d'une ligne de transmission téléphonique), il est parfois nécessaire de s'assurer de la qualité des informations transmises, qui peuvent être altérées par parasites et micro-coupures de ligne.

Le huitième bit (la parité) indiquera donc, par son état (1 ou 0) si les bits du code ASCII sont en nombre pair ou impair, ce qui permet à l'organe récepteur de vérifier la validité de chaque octet transmis (ou presque, car une erreur peut en cacher une autre... Mais il y a des moyens plus exhaustifs pour venir aussi à bout de cela...).

Soit à transmettre, par exemple, le caractère "C" qui, codé en ASCII, donne 43 H. On transmettra en réalité

| 1 | 1 0 0 0 0 1 1 |

code ASCII = 43H

parité

Si, en cours de transmission, il y a perte ou ajout de bits, il y a très peu de chances pour que le bit de parité soit conforme. L'organe récepteur le signalera donc, soit en affichant un caractère spécial, soit en demandant à l'émetteur le renouvellement de ce caractère.

Mais, là encore, c'est l'anarchie la plus complète !

Certains organes périphériques attendent une parité paire (0 si paire), d'autre une parité impaire (1 si paire), - mais dans ce cas, le bit de parité participe-t'il au calcul ou non... ? - Certains autres l'ignorent totalement (ce qui, tout compte fait, n'est peut-être pas si bête...) et bien souvent, c'est le dialogue de sourds ! Et une fois de plus, c'est le programme chargé de la transmission qui devra arrondir les angles...

Presque tous les ordinateurs possèdent une instruction magnifique et qui a un rôle parfois très important : elle s'appelle ***HALT***.

Comme son nom l'indique, elle ne fait strictement rien, et ce n'est pas toujours facile de ne rien faire !

Alors étendez-vous sur un tapis (les électrons circulent mieux ainsi), laissez aller tous vos muscles, et essayez de vous mettre en HALT pour quelques instants.

CHAPITRE 2

INTRODUCTION A L'ASSEMBLEUR

L'assembleur, selon son degré d'évolution - eh oui, lui aussi peut être plus ou moins évolué -, met à la disposition du programmeur plusieurs types de fonctions :

- Les fonctions réalisées par les instructions du microprocesseur (elles sont impératives dans tout assembleur digne de ce nom). Ce sont les ***instructions*** assembleur ;

- Les fonctions complémentaires, dont certaines ne sont présentes que sur les assembleurs plus évolués et servant, pour les unes à déterminer le début et la fin du programme (elles sont obligatoires), pour les autres à faciliter l'écriture du programme, sa modification éventuelle, et améliorer sa lisibilité et sa compréhension. Nous les appellerons ***directives*** ou ***pseudo-instructions***.

SYMBOLIQUE DE L'ASSEMBLEUR

Il existe certaines fonctionnalités implicites que tout bon assembleur se doit de posséder. Il en est ainsi, par exemple, de l'adressage ***symbolique*** qui facilite considérablement l'écriture du programme et qui consiste à utiliser des symboles alphanumériques pour désigner des adresses mémoires ou des valeurs.

Ces symboles sont appelés des ***étiquettes*** ou des ***labels***. Les labels sont donc un moyen mnémonique permettant au programmeur d'associer une adresse mémoire avec son contenu.

Ainsi, l'adresse mémoire destinée à contenir un prix par exemple, ne sera pas appelée adresse '12A4H' ... mais 'PRIX'.

Ce sera une des tâches de l'assembleur de remplacer ensuite ces labels par de véritables adresses mémoire, au moment de la génération du code machine.

Tout évolué qu'il soit, le Basic ne possède pas ce mode d'adressage. Il serait pourtant si pratique d'écrire :

GOTO DEBUT

ou

GOSUB CONVER

plutôt que l'habituel :

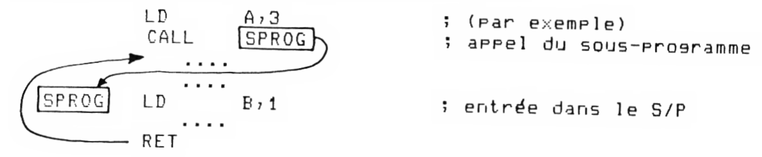
GOTO 10

ou le non moins évasif :

GOSUB 1500

qui sont d'une grande pauvreté sémantique.

Avec l'assembleur, nous pourrons écrire :



On peut, de même, spécifier la valeur d'une constante au moyen d'un symbole :

```
TOTO    EQU     4               ; TOTO prend la valeur 4
        ...
        ...
        LD      A,TOTO          ; on charge 4 dans le registre A
```

On utilise, pour ce faire, une ***directive*** de l'assembleur qui spécifie, dans cet exemple, que le symbole 'TOTO' est équivalent (EQU) à la valeur '4'.

Par la suite, l'instruction 'LD, A,TOTO' chargera cette valeur 4 dans le registre A, de la même façon que si l'on avait fait : LD A,4, mais c'est plus parlant !

FONCTIONS DE L'ASSEMBLEUR

Lorsque le code source du programme à assembler a été introduit en mémoire, en général au moyen d'un autre programme spécial appelé "éditeur de texte", ou bien de l'assembleur lui-même, qui porte alors le nom d'"éditeur-assembleur", le programmeur assembleur examine ce code-source et commence l'assemblage proprement dit, en déterminant tout d'abord la longueur de chaque instruction machine correspondant à chaque instruction source.

En effet, selon le type d'instruction, la longueur peut varier de 1 à 4 octets, du moins dans le cas du Z80. Ceci permet de déterminer l'adresse mémoire du début de chaque instruction machine à générer. Parallèlement, l'assembleur construit une "table" (ou tableau) des symboles rencontrés. Enfin, un contrôle de la syntaxe du code source est effectué en premier niveau. C'est ce que l'on nomme la PASSE 1 de l'assembleur.

Au cours de la PASSE 2 qui prend la suite, l'assembleur examine à nouveau le code source, affecte une adresse réelle à chaque symbole de la table, et commence à générer le code objet ainsi que le listing d'assemblage, et procède à une vérification des erreurs de second niveau n'ayant pu être détectées en première passe.

Le listing comprendra, outre le reflet du code source, les adresses mémoire et le code objet généré, ainsi, bien entendu, que les erreurs éventuelles reconnues par l'assembleur.

Avant de voir un exemple de listing d'assemblage, il nous faut étudier la forme et la syntaxe du code source.

FORMAT DU CODE SOURCE

Selon notre habitude, nous nous baserons sur la syntaxe de l'assembleur Z80 de Zilog, sachant que la compréhension de ses principes fondamentaux facilitera grandement le passage éventuel à un autre type de microprocesseur.

Le code source est formé de lignes source écrites les unes à la suite des autres. Chaque ligne source possède deux types de format :

- Le format commentaire, commençant en général par un caractère spécial (';' en Z80 et 8080) est destiné à documenter le programme, et n'est pas traité par l'assembleur qui se contente de le transmettre au listing, sans autre modalité.

```
; CECI EST UN COMMENTAIRE
```

- Le format fonctionnel qui est traité par l'assembleur et se compose de quatre champs : le champ LABEL, le champ OPERATION, le champ OPERANDE, le champ COMMENTAIRE.

Exemple :

```
LABEL   OPERATION      OPERANDE       COMMENTAIRE
-----   ---------      --------       ----;------
DEBUT   LD             A,1            ; CHARGE 1 DANS A
```

Selon le type de fonctions, certains champs peuvent être omis, et dans certains assembleurs le format commentaire peut être considéré comme un format fonctionnel dans lequel les champs label, opération et opérande sont omis.

Examinons maintenant les règles syntaxiques générales appliquées à ces différents champs.

LE CHAMP LABEL

Le champ label doit commencer en première position de la ligne source, par un caractère non numérique. Les caractères spéciaux (% $ = ",...) n'y sont pas toujours admis.

Il ne doit pas correspondre à un nom d'instruction, de directive ou de registre.

Il est formé de 1 à 6 (parfois 8) caractères sans espace. Dans certains assembleurs (8080 d'Intel, notamment), le label doit se terminer, au moment de sa définition, par le caractère spécial ":".

Ce champ est, bien entendu, ***optionnel***.

Exemples de labels corrects :

```
DEBUT
A2
A.VAL            (pas toujours accepté...)
VALEUR
```

Exemples de labels incorrects :

```
2A
LD               (si ce nom est celui d'une instruction)
A VAL
TROPLONG
```

LE CHAMP OPERATION

Le champ opération commence après le champ label (si présent) duquel il doit être séparé par au moins un espace.

Il est ***toujours obligatoire*** sauf dans le cas d'un format commentaire, et est formé d'un nom d'instruction ou de directive assembleur.

Exemples :

```
LD   }
PUSH }           instructions Z80
CALL }
EQU              directive assembleur
```

LE CHAMP OPERANDE

Le champ opérande est le plus compliqué des quatre... sauf dans le cas où il est absent, ce qui peut arriver parfois (HALT par exemple).

Ce champ étant intimement lié au type d'opération, nous rencontrerons deux groupes importants :

- les opérateurs monadiques (un seul argument),
- les opérateurs dyadiques (deux arguments séparés par un caractère spécial (généralement la virgule).

Dans ce dernier groupe, le premier argument représente toujours une destination, et le second une source.

Par exemple :

```
LD      A,B
```

exécute le transfert du registre B (source) dans le registre A (destination).

```
PUSH    HL      ; operande à un seul argument
LD      A,41H   ; operande à deux arguments
```

Chaque argument peut prendre à son tour l'une des significations suivantes :

1) un nom de registre,
2) un pointeur registre,
3) une donnée,
4) une adresse
5) un pointeur adresse
6) un mot-clé condition.

Bien entendu, et comme nous l'avons dit précédemment, le nombre et le type d'arguments sera étroitement lié au type de l'opération associée.

Avant d'étudier plus en détails les différentes significations de l'argument d'opérande, nous devons parler de ce que l'on appelle une expression.

Les expressions

L'un des exemples précédents nous a montré que l'on pouvait écrire :

```
LD      A,TOTO   ; pourquoi pas?
```

sachant que le label 'TOTO' avait été défini par :

```
TOTO   EQU     4
```

De même, nous pouvions écrire :

```
CALL    SPROG
```

sachant que 'SPROG' représentait symboliquement une adresse mémoire, définie ailleurs dans le programme.

Eh bien nous pouvons, maintenant, compliquer un peu la "sauce" en écrivant :

```
LD      A,TOTO+2
```

et

```
CALL    SPROG+12
```

Evidemment, les effets ne seront plus les mêmes que par le passé, mais dans le premier cas, nous chargerons la valeur 6 dans le registre A, et dans le second, l'entrée du sous-

programme ne sera plus SPROG, mais se situera 12 octets mémoire plus loin.

Nous pouvons également faire :

```
CALL    SPROG+TOTO
```

ce n'est pas interdit, et même :

```
CALL    SPROG+TOTO-2
```

le tout est de savoir exactement ce que l'on fait !

Une expression - car c'est bien d'elle dont il s'agit - est donc un ensemble de symboles et de valeurs numériques, séparés entre eux par des opérateurs algébriques et/ou logiques.

Les opérateurs algébriques, vous les connaissez déjà par le Basic. Ce sont :

+ pour l'addition
- pour la soustraction
* pour la multiplication
/ pour la division.

Bon. Et les opérateurs logiques ? De la même manière que les opérateurs algébriques font des opérations algébriques, les opérateurs logiques... Voila ! Vous avez deviné ! Si vous vous sentez pris d'un léger malaise, n'hésitez pas à abuser des annexes situées en fin de volume...

Les opérateurs logiques les plus courants dans l'assembleur sont :

- le AND (opérateur logique "ET")
- le OR (opérateur logique "OU")
- le NOT (complémentation logique)
- le XOR ("OU" exclusif)
- le MOD (ou modulo, donnant le reste de la division entière de deux valeurs)

- le SHR
- le SHL

(qui exécutent respectivement un décalage à droite 'Shift Right' et à gauche 'Shift Left' selon un nombre de bits spécifié)

- le HIGH
- le LOW

(qui font l'extraction haute 'High' ou basse 'Low' d'une valeur de 16 bits

Notons que ces deux derniers opérateurs peuvent être substitués par une combinaison de SHR et de AND.

Soit VAL une valeur de 16 bits.

```
VAL SHR 8       ---> HIGH
VAL AND OFFH    ---> LOW
```

Exemples d'emploi d'opérateurs logiques :

```
VAL1  EQU   1234H
MASK  EQU   0FBH
      LD    A,VAL1 AND 0FFH
      LD    BC,VAL1 SHR 1 + 1
      LD    DE,VAL1 OR MASK
```

Après exécution, le registre A contiendra :

```
|0 0 0 1,0 0 1 0,0 0 1 1,0 1 0 0|   1234H
                |1 1 1 1,1 1 1 1|   0FFH
                |0 0 1 1,0 1 0 0|   A = 34H
```

le registre BC contiendra :

```
|0 0 0 1,0 0 1 0,0 0 1 1,0 1 0 0|   1234H
|0 0 0 0,1 0 0 1,0 0 0 1,1 0 1 0|   1234H SHR 1
|0 0 0 0,0 0 0 0,0 0 0 0,0 0 0 1|   +1
|0 0 0 0,1 0 0 1,0 0 0 1,1 0 1 1|   BC = 091BH
```

le registre DE contiendra :

```
|0 0 0 1,0 0 1 0,0 0 1 1,0 1 0 0|   1234H
                |1 1 1 1,1 0 1 1|   0FBH
|0 0 0 1,0 0 1 0,1 1 1 1,1 1 1 1|   BC = 12FFH
```

LES SIGNIFICATIONS DE L'ARGUMENT D'OPERANDE

Voyons maintenant les différentes significations pouvant être prises par l'argument d'opérande.

1.Un nom de registre

Pas de problème ici. Il s'agira de l'un des registres (simple ou double) connu du microprocesseur :

A, B, C, D, E, H, L, I et R pour les simples

AF, BC, DE, HL, SP, IX et IY pour les doubles.

Il est à noter au passage que le registre F ne peut être utilisé comme simple registre et doit être traité comme la partie faible du double registre AF.

Exemples :

```
PUSH   HL
LD     A,3
LD     B,A
```

2. Un pointeur registre

Il est formé d'un nom de double registre, encadré par les parenthèses :

(BC), (DE), (HL), (SP), (IX) et (IY).

Si AF ne figure pas à la liste, il y a une raison : ces doubles registres doivent contenir, dans ce mode, une adresse mémoire de 16 bits et le registre F ne peut être utilisé de cette façon. Nous verrons pourquoi un peu plus loin.

Ce qui nous intéresse dans le pointeur de registre, ce n'est pas le contenu même du double registre, mais la donnée mémoire située à l'adresse pointée par ce double registre.

Ainsi, après l'exécution de :

```
LD      HL,1234H        ; charge adresse dans HL
LD      A,(HL)          ; lecture par pointeur registre
```

le registre A contiendra l'octet mémoire situé à l'adresse 1234H.

Le cas de IX et IY est un peu plus complexe, mais pas de panique ! Nous en viendrons à bout !

Ces registres sont, en effet, utilisés par le mode d'adressage dit "indexé", et leurs pointeurs registres admettent une expression du genre :

```
(IX+2) ou (IX-VAL)
```

Nous en reparlerons lorsque nous aborderons les différents modes d'adressage.

3. Une donnée

Selon le type d'instruction, il s'agira d'une donnée 8 ou 16 bits, prise sous sa forme numérique ou symbolique, ou encore sous la forme d'une expression.

Exemples :

```
LD      A,2      ; forme numerique 8 bits
LD      A,VAL    ; forme symbolique
LD      A,VAL-2  ; forme expression
LD      HL,ADR   ; donnee 16 bits
```

4. Une adresse

Comme il s'agit d'une adresse mémoire, elle sera toujours sur 16 bits et ne pourra, de ce fait, être contenue que dans les doubles registres.

Les remarques concernant la donnée 16 bits s'appliquent ici également :

```
LD      BC,ADRESS
LD      SP,PILE+VAL-2
LD      HL,1234H+2
```

5. Un pointeur adresse

De même que pour le pointeur registre nous accédions à l'octet mémoire au travers d'un double registre, nous pourrons aussi y accéder... tout bonnement au travers de son adresse mémoire, sous forme numérique, symbolique ou d'expression :

```
LD      A,(1234H)
```

charge dans le registre A l'octet situé à l'adresse mémoire 1234H.

```
LD      BC,(SPROG+2)
```

charge dans le double registre BC les deux octets situés respectivement aux adresses mémoire SPROG+2 et SPROG+3.

Attention toutefois : tous les coups ne sont pas permis ! L'instruction :

```
LD      B,(... )
```

n'existe pas ...!

6. Un mot-clé de condition

Oh ! Rien de bien compliqué, rassurez-vous !

Vous savez qu'en Basic il est possible d'opérer certains branchements (GOTO ou GOSUB) en fonction de l'établissement de certaines conditions :

```
IF A < 0 THEN GOTO ...
IF B >= 4 THEN GOSUB ...
```

Les instructions machines des microprocesseurs peuvent également tester la présence ou l'absence de conditions matérialisées par des ***indicateurs*** (***flags*** = drapeaux en Anglais).

Dans le Z80, ces indicateurs occupent les bits du registre F (F comme flags) et peuvent être interrogés à tout instant. A chaque condition correspond un mot-clé reconnu par l'assembleur, et qui figure toujours en premier argument de l'opérande des instructions de branchement.

Ces mots-clés sont :

NZ non zéro,
Z zéro,
NC non carry,
C carry
PO parité impaire (Odd),

PE parité paire (Even),
P plus, positif,
M moins, négatif,

Exemple :

```
JP      Z,SPROG
LD      A,1
....    ...
```

ce qui signifie : si l'indicateur Z (zéro) est positionné (bit = 1) le programme doit se poursuivre (instruction JP = Jump, saut en français) à l'adresse SPROG, sinon l'instruction suivante est exécutée (LD A,1).

Nous verrons la signification de ces différents indicateurs plus tard, c'est promis !

DEFINITION DES DONNEES

Revenons aux données, et parlons des moyens pouvant être utilisés pour les définir avec l'assembleur, dans la mesure, il va sans dire, où celui-ci le permet.

Une donnée (constante ou adresse) peut être spécifiée :

1. *En notation décimale signée :*

```
LD      HL,64200
LD      A,255
LD      B,-3
```

2. *En notation hexadécimale*

```
LD      A,0FFH
LD      HL,9A00H
```

Notons au passage la présence du zéro devant la lettre hexadécimale, qui indiquera à l'assembleur qu'il ne s'agit point d'un label appelé FFH (pourquoi pas ?) mais d'une valeur numérique. En effet, les labels ne commencent jamais par un chiffre.

3. *En notation octale* (base 8, moins utilisée) :

```
LD      A,640
```

ou, plus clairement (la lettre O est souvent confondue avec le chiffre 0) :

```
LD      A,64Q
```

Il y a quelques années, le système de numération octal (base 8) était d'un usage courant sur les ordinateurs de l'époque. Comme tout change, le système hexadécimal (base 16) a pris la place. Néanmoins, il reste encore certains matériels (et certains programmeurs) qui persistent à utiliser l'ancien système.

Les suffixes O et Q sont pour eux.

4. *en notation binaire*

```
LD      A,11000101B
```

5. *Sous forme d'un caractère ASCII :*

```
LD      A,'A'
LD      B,'*'
```

ce qui est plus parlant que :

```
LD      A,41H
LD      B,2AH
```

6. *Sous forme d'un symbole ou label :*

```
LD      A,VAL

CALL    SPROG
```

nécessairement défini dans le programme, devons-nous le rappeler.

7. *Sous forme d'une expression :*

```
LD      A,2*(VAL-2)+TOTO AND 4
```

Les expressions faisant référence aux adresses peuvent également utiliser le symbole spécial '$' qui prend la valeur de l'adresse début de l'instruction. Par exemple :

```
Adresse             Instruction
-------             -----------
1000H               CALL     $+6
  .                      .
  .                      .
  .                      .
1006H               LD       HL,$-2
```

Le CALL provoquera un branchement à l'adresse actuelle du début de l'instruction (1000H) augmenté de 6, soit 1006H. De même, HL contiendra la valeur 1006H-2, soit 1004H.

EXEMPLE DE PROGRAMME

Tout ceci paraît peut-être un peu confus, mais les idées devraient se remettre en place après examen du listing suivant que nous allons commenter ligne par ligne.

Certaines instructions ne vous seront pas toujours familières, mais il est inutile d'invoquer ce prétexte pour abandonner en si bon chemin...!

```
                 00100 ;-----------------------------------;
                 00110 ; EXEMPLE DE PROGRAMME REALISANT UNE  ;
                 00120 ; PERMUTATION DE DEUX ZONES MEMOIRE   ;
                 00130 ;-----------------------------------;
                 00140 ;
8000             00150          ORG     8000H          ; ORIGINE
                 00160 ;
                 00170 ; DECLARATION D'ADRESSES EXTERIEURES
                 00180 ;
9000             00190 BUFF1    EQU     9000H          ; BUFF 1
90C8             00200 BUFF2    EQU     BUFF1+200      ; BUFF 2
                 00210 ;
                 00220 ; ENTREE DU PROGRAMME
                 00230 ;
8000 31FFFF      00240 DEBUT    LD      SP,0FFFFH      ; PILE
8003 210090      00250          LD      HL,BUFF1       ;
8006 11C890      00260          LD      DE,BUFF2       ;
8009 06C8        00270          LD      B,200          ;
800B CD1C80      00280          CALL    PERMUT         ; APPEL 1
800E 21C890      00290          LD      HL,BUFF2       ;
8011 110090      00300          LD      DE,BUFF1       ;
8014 06C8        00310          LD      B,200          ;
8016 CD1C80      00320          CALL    PERMUT         ; APPEL 2
8019 C31980      00330 ARRET    JP      ARRET          ; BOUCLE
                 00340 ;
                 00350 ; ROUTINE DE PERMUTATION
                 00360 ; EN ENTREE: HL POINTE SUR LE 1ER BUFFER
                 00370 ;            DE POINTE SUR LE 2EME BUFFER
                 00380 ;            B CONTIENT LA LONGUEUR
                 00390 ; EN SORTIE: HL = ADR BUFFER 1 + LONGUEUR
                 00400 ;            DE = ADR BUFFER 2 + LONGUEUR
                 00410 ;            B = 0
                 00420 ;            C MODIFIE PAR LA ROUTINE
                 00430 ;
801C 4E          00440 PERMUT   LD      C,(HL)         ; LECTURE
801D 1A          00450          LD      A,(DE)         ; LECTURE
801E 77          00460          LD      (HL),A         ; ECRIT.
801F 79          00470          LD      A,C            ; C --> A
8020 12          00480          LD      (DE),A         ; ECRIT.
8021 23          00490          INC     HL             ; ADR+1
8022 13          00500          INC     DE             ; ADR+1
8023 05          00510          DEC     B              ; LG-1
8024 C21C80      00520          JP      NZ,PERMUT      ; SI # 0
8027 C9          00530          RET                    ; SI = 0
                 00540 ;-------------------------------;
8000             00550          END     DEBUT
00000 TOTAL ERRORS


ARRET  8019 00330   00330
BUFF1  9000 00190   00200 00250 00300
BUFF2  90C8 00200   00260 00290
DEBUT  8000 00240   00550
PERMUT 801C 00440   00280 00320 00520
```

BUT DU PROGRAMME

Il s'agit de permuter deux zones mémoires en appelant une routine appropriée, puis de les permuter de nouveau, afin de retrouver la configuration initiale, par un second appel de cette routine.

Ce programme peut être résolu plus simplement, mais en utilisant des instructions plus complexes.

Essayez déjà de le comprendre tel qu'il est. Vous pourrez l'améliorer plus tard...

Nota : Ce programme a été réalisé avec l'éditeur-assembleur du TRS-80.

EXPLICATION DU PROGRAMME, ligne par ligne (ou presque !)

Nous pouvons déjà, au premier coup d'oeil, déterminer deux zones principales sur le listing :

- Une *zone contenant le code source* tel qu'il a été introduit au moyen de l'éditeur de texte. En général, ce dernier numérote automatiquement chaque ligne source au moment de son introduction (100, 110...).

- Une *zone générée par l'assembleur* (à gauche) et comprenant deux colonnes : une pour les adresses, l'autre pour le code machine généré.

LIGNES 100 à 140
Ce sont des lignes commentaires, commençant par le caractère spécial ';'.

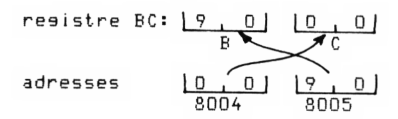
LIGNE 150
Cette directive ou pseudo-instruction de l'assembleur est utilisée pour fixer l'adresse d'origine du programme en mémoire (là où il sera implanté). Dans l'exemple, l'adresse sera 8000H et nous voyons, en effet, que la première instruction du programme (LD SP,...) sera bien à cette adresse.

LIGNES 190 et 200
Les deux zones mémoire utilisées (appelées ***buffers***) étant extérieures au programme, il est nécessaire de définir leurs adresses au moyen de la directive déjà entrevue : EQU. Le deuxième buffer peut être défini relativement au premier, comme le montre la ligne 200, en spécifiant que son adresse début sera égale à l'adresse début du premier buffer, augmentée de sa longueur. A gauche de la ligne, nous pouvons constater, non sans une certaine émotion... que l'assembleur a compris, et qu'il a généré l'adresse 90C8H, qui est bien égale à 9000H + 200. Si, par la suite, on veut changer l'adresse des deux buffers, il suffira simplement de modifier la ligne 190. Notez enfin que ces directives ne génèrent pas de code machine.

LIGNE 240
C'est la première instruction du programme dans toute sa splendeur, avec un champ label (DEBUT), un champ instruction (LD), un champ opérande (SP,0FFFFH) et un champ commentaire

(;PILE). Elle stipule que le registre pointeur de pile 'pointera' en fin de mémoire (dernière adresse = FFFF), ce qui est nécessaire, puisque nous utiliserons la pile, par la suite, avec les instructions CALL. A gauche de la ligne, l'assembleur a généré l'adresse mémoire (8000H) qui est toujours en hexadécimal, et le code machine correspondant à l'instruction (31FFFF). Cette instruction occupe trois octets, l'adresse mémoire de l'instruction suivante sera donc 8003H.
Nous pouvons deviner, en passant, que le code machine contient la codification de l'instruction (31) et un opérande (FFFF). Reportez-vous aux annexes pour les différents codes instructions).

LIGNE 250
Le double registre HL est chargé avec l'adresse début du buffer 1. Le code machine généré est formé du code instruction (21) et de l'opérande (9000) qui correspond à l'adresse définie plus haut par un EQU. Bizzarement, nous constatons que cette valeur a été "renversée" dans le code machine généré par l'assembleur. C'est normal : le registre "faible" C est chargé par l'adresse "faible" 8004H et le registre "fort" B par l'adresse "forte" 8005H qui contient bien 90H...



Le ';' du champ commentaire n'est présent que pour "faire joli" !

LIGNE 260
Même opération, mais avec l'adresse début du deuxième buffer, qui sera chargée dans le registre DE. Notons que ces deux dernières instructions auraient pu être écrites :

```
LD      HL,9000H
LD      DE,90C8H
```

mais pour les mêmes raisons que celles exposées plus haut, si l'on décide de déplacer les buffers, il faudrait alors rectifier tous les opérandes relatifs à ces adresses alors que, dans notre cas, il suffit de modifier une seule ligne, la 190. Et puis... cela nous évite aussi d'avoir à calculer la somme 9000H + 200 !

LIGNE 270
La longueur des buffers fixée à 200 octets est chargée dans le registre B. Là, par contre, nous aurions pu écrire :

```
LD      B,LONG
```

avec :

```
LONG    EQU     200     ; longueur = 200 octets
```

ainsi que :

```
BUFF2   EQU     BUFF1+LONG
```

Ce type d'instruction comporte deux octets, comme on peut le voir dans la zone du code machine générée par l'assembleur.

LIGNE 280
C'est l'instruction CALL, bien connue maintenant. Nous pouvons nous apercevoir que l'assembleur a bien généré, dans le code machine, l'adresse réelle correspondant au label PERMUT, soit 801CH (inversée...). Il n'est pas bête, hein ?

LIGNES 290 à 320
Même séquence que précédemment, mais les deux buffers sont permutés de nouveau. Nous aurions pu écrire :

```
LD      HL,BUFF1
LD      DE,BUFF2
```

ce qui n'aurait d'ailleurs rien changé du tout.

LIGNE 330
Là, nous entrons dans un cercle vicieux ! Lorsque le microprocesseur exécute cette instruction, il saute (JP = JUMP) à l'adresse indiquée par l'opérande qui se trouve être, dans le cas présent... l'adresse de cette même instruction ! Nous aurions pu écrire également :

```
JP      $
```

vous en souvenez-vous ?

Nous abordons maintenant la routine PERMUT appelée deux fois dans le programme principal. Sans les instructions CALL/RET, il aurait fallu écrire deux fois cette séquence d'instructions...!

LIGNE 440
C'est le point d'entrée de la routine. L'instruction LD C,(HL) possède un opérande de type "pointeur registre" et correspond à un seul octet de code machine. Le registre C (premier argument de l'opérande = destination) est chargé avec l'octet pointé par l'adresse mémoire stockée dans le double registre HL (deuxième argument = source). Cet octet appartient au buffer 1.

LIGNE 450
L'octet du buffer 2 est, à son tour, chargé dans le registre A...

LIGNE 460
...et est écrit dans le buffer 1 pointé par HL. Notez le sens des arguments : (HL) = premier argument = destination, A = deuxième argument = source.

LIGNE 470
Comme l'instruction :

```
LD      (DE),C
```

n'existe pas... il faut bien se faire une raison, et transférer le contenu du registre C dans le registre A...

LIGNE 480
... puis l'écrire dans le buffer 2, toujours pointé par DE.

LIGNES 490 et 500
Ces deux nouvelles instructions exécutent une addition de 1 sur le double registre spécifié en opérande (incrément). HL pointera alors sur l'octet suivant du buffer 2.

LIGNE 510
Ici, c'est l'opération inverse (décrement) qui est effectuée sur le registre B, lequel, à l'origine, contenait 200, et contiendra après exécution de cette instruction, 199.

LIGNES 520 et 530
A force de décrementer le registre B, il finira fatalement par atteindre une valeur nulle. Dans ce dernier cas, le "drapeau" indicateur Z sera "monté" par le microprocesseur. Le but de l'instruction JP NZ est justement de tester la présence de cet indicateur : s'il est non-zéro (NZ), un saut est effectué à l'adresse donnée en second argument de l'opérande (début de la routine PERMUT, dans notre cas). Dans le cas contraire (tout le buffer a été transféré) le saut n'a pas lieu, et l'exécution se poursuit par l'instruction suivante, qui est un RET dans notre cas. Il y a donc retour au programme principal.

LIGNE 550
Fin du programme signalée par la pseudo-instruction END (pas de code généré) précisant en outre que le programme devra être lancé à l'adresse DEBUT.

A ce niveau là, il faut éclaircir un petit point. Il y a une légère différence entre les appellations code machine et code objet.

Le **code machine** est celui figurant dans la deuxième colonne, à gauche du listing, et correspond aux instructions et aux données traitées par le microprocesseur. C'est ce code qui sera intégralement transféré en mémoire, lors du chargement du programme.

Le **code objet** est le code généré par l'assembleur. Ce code est en général transmis à un organe périphérique (magnétophone, disquette) qui en assurera la sauvegarde (mémorisation). Bien entendu, le code objet contient le code machine, mais aussi certaines informations relatives à l'implantation mémoire du programme, sa longueur et son adresse de lancement, définie justement par la directive END.

Ces informations ne sont pas chargées en mémoire avec le code machine, mais sont destinées à informer le programme de chargement appelé ***chargeur***.

En effet, le seul examen du code machine ne permet pas de dire, a priori, quelles adresses mémoire le programme doit occuper, et quelle doit être la première instruction à exécuter (qui n'est pas forcément en première position du code machine). Nous verrons cela un peu plus tard...

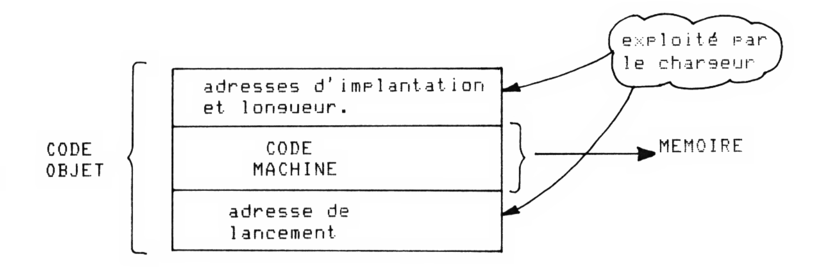


FIGURE 9

Revenons à notre listing.

Après nous avoir signalé que le programme ne comportait pas moins de "0 erreurs", l'assembleur fournit, en prime, une liste des labels triés, donnant les adresses de définition de ces labels, les numéros de lignes correspondants, ainsi que ceux où ils sont appelés.

Nous voyons, par exemple, que le label BUFF2 correspond à l'adresse 90C8H, est définie en ligne 200 et est utilisée aux lignes 260 et 290. Que demander de plus ?

Voici un autre listing du même programme (enfin presque...), mais dans lequel quelques erreurs se sont glissées.

A vous d'en trouver les causes.

```
                 00100 ;----------------------------------------;
                 00110 ; VOICI UN EXEMPLE DE PROGRAMME A NE      ;
                 00120 ; PAS IMITER... ET CE N'EST PAS UNE      ;
                 00130 ; QUESTION DE COPYRIGHT...               ;
                 00140 ;----------------------------------------;
UNDEFINED SYMBOL
0000             00160          ORG     FFFFH
BAD LABEL                                          Aïe!
00170 DECLARATIONS D'ADRESSES EXTERIEURES
2328             00180 BUFF1    EQU     9000
UNDEFINED SYMBOL
00C8             00190 BUFF2    EQU     BUFF+200
                 00200 ;
                 00210 ; ENTREE DU PROGRAMME
ILLEGAL OPCODE                                        HOUPSS...!
00220 DEBUT      LOAD     SP,0FFFFH
ILLEGAL ADDRESSING MODE
00230            LD       HD,BUFF1
0000 11C800      00240          LD      DE,BUFF2
FIELD OVERFLOW
0003 06D0        00250          LD      B,2000
ILLEGAL OPCODE
00260            COLL     PERMUT
0005 21C800      00270          LD      HL,BUFF2
0008 112823      00280          LD      DE,BUFF1
ILLEGAL ADDRESSING MODE
00290            LD       B.200
000B CD1100      00300          CALL    PERMUT
000E C30E00      00310          JP      $
                 00320 ;
                 00330 ; ROUTINE
BAD LABEL
00340 ,
0011 4E          00350 PERMUT   LD      C,(HL)
ILLEGAL ADDRESSING MODE                                  GLouP!
00380            LD       (DE),C
0012 24          00390          INC     H
0013 C9          00400          RET
NO END STATEMENT
00011 TOTAL ERRORS                   KÏ...KÏ...KÏ!


BUFF    0000 UNDEF***00190
BUFF1   2328 00180     00230 00280
BUFF2   00C8 00190     00240 00270      GRRR....!
FFFFH   0000 UNDEF***00160 00220
PERMUT  0011 00350     00260 00300
```

LES INSTRUCTIONS NECESSAIRES A L'ASSEMBLEUR

Avant de nous lancer dans l'écriture d'un programme en assembleur, il paraît utile de prendre connaissance des fonctions pouvant être réalisées par les instructions du microprocesseur. Un minimum est nécessaire, mais le Z80 n'est pas "juste" de ce côté là !

Pour cela, il n'y a pas de secret : il faut consulter la table des instructions fournie par le constructeur du microprocesseur, et essayer de les mémoriser. En pratiquant un peu, cela vient tout seul.

Nous allons, malgré tout, tenter un découpage des différentes fonctions.

Il faut, tout d'abord, pouvoir dialoguer avec la mémoire, c'est la moindre des choses ! Ce dialogue s'établira entre une référence mémoire et un registre, au moyen des instructions LD (load = charger en anglais) qui permettent le dialogue dans les deux sens :

```
registre ---> memoire   (ecriture memoire)
memoire  ---> registre  (lecture memoire)
```

Le dialogue est souvent nécessaire également entre les registres eux-mêmes. L'instruction LD le permet aussi.

Il faut aussi disposer d'un minimum de fonctions arithmétiques. Pour tous les microprocesseurs 8 bits actuels, ce sera toujours un minimum dans ce domaine : addition et soustraction. Un point c'est tout.

"*Pardon ?*"... direz-vous... "*j'ai mal compris ?*".

Non. C'est dur, mais il faut bien se rendre à l'évidence. Si l'on veut faire une multiplication ou une division en langage machine... il faut "se la programmer". Rassurez-vous toutefois : il existe d'excellentes routines toutes faites...
Les instructions de comparaison peuvent aussi être classées dans ce groupe. En général, elles comparent toujours "quelque chose" avec le registre A. Il faudra s'y faire.

Les instructions réalisant les opérations logiques doivent aussi être présentes : ET, OU, OU exclusif, NON, décalages, sont très utilisés dans les programmes assembleur, vous vous en rendrez compte.

Les instructions de commande et de contrôle internes, qui agissent directement sur certaines fonctions intrinsèques au microprocesseur, peuvent aussi avoir une utilité.

Les instructions de "rupture de séquence", ou instructions de saut (conditionnel ou non), sont parfaitement indispensables. Y sont incluses les instructions d'appel et de retour de sous-programmes.

Les instructions d'entrée/sortie permettent le dialogue avec les périphériques. Sans elles, toute communication avec le monde extérieur serait impossible. La programmation d'un

ordinateur serait exclue, et vous ne seriez pas en train de lire ce livre...

Les instructions de pile ne sont pas indispensables, mais elles contribuent grandement à l'efficacité du programme.

Les instructions spéciales opérant sur bits ou sur chaînes de caractères, sans être également indispensables, sont bienvenues lorsqu'elles sont présentes.

Mais il n'y a pas que les instructions qui comptent. Il y a aussi la façon de s'en servir.

LES MODES D'ADRESSAGE

Qu'appelle-t-on exactement modes d'adressage ? Nous pouvons dire que ce sont les différents moyens d'accéder à une donnée, tout simplement.

Les modes d'adressage jouent un très grand rôle dans la puissance, la maniabilité, la souplesse et l'efficacité du jeu d'instructions d'un ordinateur. Ils facilitent et simplifient l'écriture du programme, tout en améliorant sa vitesse d'exécution.

Le Z80 possède 7 modes d'adressage que nous allons examiner maintenant.

Adressage immédiat

Dans ce mode, la valeur à charger se trouve en zone opérande de l'instruction et est donc *immédiat*ement accessible.

```
LD      A,1
```

Adressage registre

En partant de l'hypothèse que la valeur à charger se trouve déjà, par exemple, dans le registre B, nous utiliserons ce mode d'adressage en exécutant l'instruction :

```
LD      A,B
```

Adressage indirect registre

C'est le pointeur registre évoqué précédemment. Si la valeur 1 se trouve à l'adresse mémoire 1234H, et que le double registre HL contient cette adresse (LD HL,1234H), l'instruction :

```
LD      A,(HL)
```

chargera la valeur 1 dans le registre A, en passant indirectement par le registre HL.

Adressage direct

En reprenant l'exemple précédent, et en faisant :

```
LD      A,(1234H)
```

nous chargerons directement la valeur 1 dans A. D'où le nom d'adressage direct.

Adressage relatif

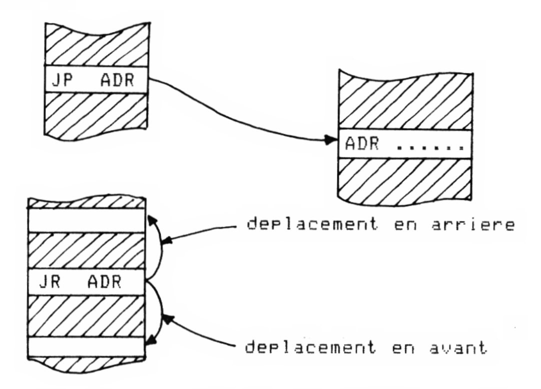
Voici un mode d'adressage dont nous n'avons encore jamais parlé. Il est utilisé uniquement par les instructions de saut.

Le principe en est extrêmement simple. Nous avons vu dans le listing du programme de permutation de zones, que l'instruction JP NZ occupait trois octets de code machine : 1 pour le code instruction et 2 pour l'argument d'opérande contenant une adresse de 16 bits.

Nous aurions pu remplacer cette instruction de saut par :

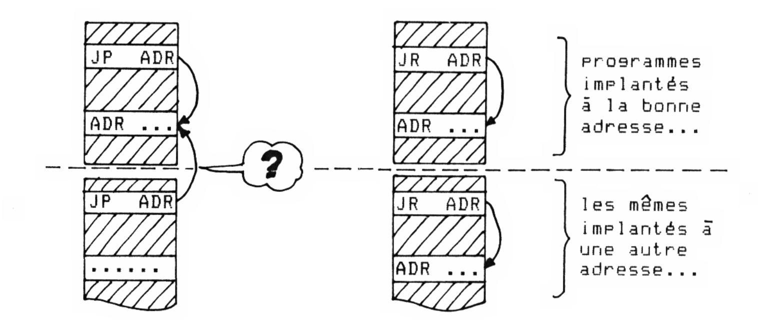
```
JR      NZ,PERMUT
```

et, dans ce cas, le code généré n'aurait occupé que 2 octets seulement, l'un pour le code opération (qui ne sera pas le même, bien sûr) et l'autre donnant, non pas une adresse mais un déplacement relatif. Il y a en effet deux moyens pour "sauter" :



Dans ce dernier cas, l'opérande généré en code machine par l'assembleur est égal à la différence entre l'adresse à atteindre et l'adresse actuelle. Il y a toutefois une restriction : le déplacement étant fixé sur un octet, il ne pourra être supérieur à 127 (en avant) et à 128 (en arrière).

Il y a donc avantage à utiliser les branchements relatifs lorsque cela est possible, d'autant plus que le programme est ainsi rendu translatable, c'est-à-dire qu'il peut s'exécuter sans problème à différents emplacements mémoire, à condition toutefois de n'utiliser que des branchements relatifs à l'intérieur du programme, ce qui exclut d'office l'instruction CALL qui n'a, malheureusement, pas d'équivalent en adressage relatif.



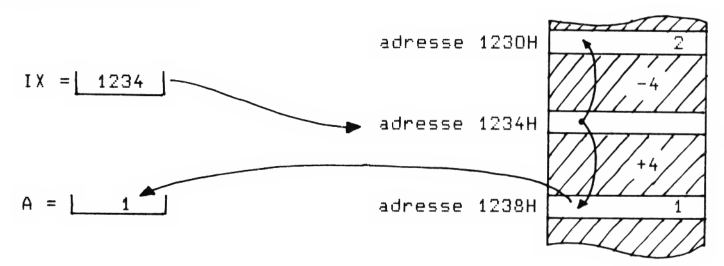
Adressage indexé

En réalité, il s'agit plus exactement d'un adressage pseudo-indexé. Nous verrons plus loin ce qu'il convient d'appeler adressage indexé.

Il s'agit ici d'accéder à une donnée par un déplacement -positif ou négatif- relatif à une adresse contenue dans l'un des registres IX et IY réservés à cet effet.

```
LD      A,(IX+4)
```

Dans cet exemple, le registre A sera chargé par l'octet mémoire situé à l'adresse stockée dans le registre IX, augmentée de 4 :



et si l'on avait fait :

```
LD      A,(IX-4)
```

c'est la valeur 2 qui aurait été chargée dans A.

Comme pour les branchements relatifs, le déplacement est limité à 127 octets en avant et à 128 en arrière.

Adressage bit

Ce mode intéressant permet de lire, écrire, tester, non plus un octet mais un bit bien précis de cet octet, qui pourra être

soit dans un registre, soit en mémoire.

```
SET     0,A
```

positionne à la valeur "1" le bit numéro zéro du registre A.

```
A = |_|_|_|_|_|_|_|1|
     7 6 5 4 3 2 1 0
```

les autres bits n'étant, évidemment, pas modifiés.

D'AUTRES MODES D'ADRESSAGE

Oui, il existe encore d'autres modes d'adressage, mais présents sur les "minis" et les "gros" ordinateurs seulement. Parmi les plus usuels, nous citerons rapidement :

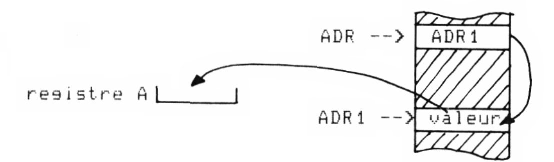
L'adressage indirect mémoire

Nous connaissions déjà l'adressage indirect registre, mais dans le cas présent, l'adresse, plutôt que d'être stockée dans un registre, se trouve en mémoire.

Par exemple, le signe "@" indique l'opération d'indirection :

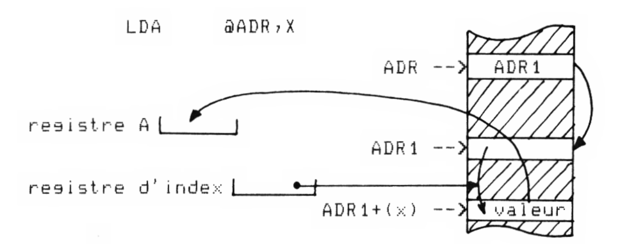
```
LDA     @ADR
```

chargera dans le registre A, non pas la donnée située à l'adresse ADR, mais celle située à l'adresse contenue dans ADR. Vous voyez la nuance ? C'est un double adressage à arbre à cames en tête...



L'adressage indirect indexé

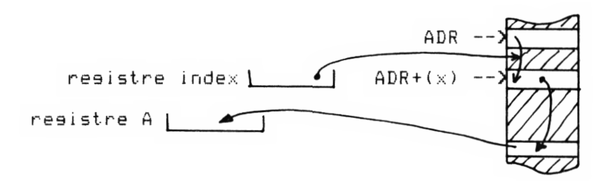
Le principe est le même que ci-dessus, mais à l'adresse obtenue, on ajoute le contenu du registre d'index (X en principe).



Encore faut-il distinguer dans ce mode l'adressage post-indexé de l'adressage pré-indexé.

Celui décrit ci-dessus est le post-indexé, car on ajoute le contenu du registre d'index en dernière opération.

Dans le mode pré-indexé, cette opération s'effectue avant :



L'adressage basé

Ce mode d'adressage permet la translatabilité intégrale des programmes, telle que nous l'avons évoquée dans l'adressage relatif (mais sans ces petites restrictions...).

Pour pouvoir utiliser ce mode, l'ordinateur (le microprocesseur) doit être muni d'un registre de base dont le contenu est systématiquement ajouté aux adresses mémoire appelées dans le programme.

Bien entendu, le jeu d'instructions doit "connaître" ce registre.

```
LDA     B.ADR
JMP     B.DEBUT
```

Pour B = 0, c'est le mode normal que nous connaissons. Mais il y a 65535 autres possibilités d'implantation... (Pour un registre de base de 16 bits).

ROLE DES REGISTRES

Nous allons maintenant passer en revue les différents registres de Z80 en parlant un peu des fonctions qu'ils occupent, car certains, nous allons le voir sous peu, sont spécialisés, et il ne s'agit pas de les utiliser à tort et à travers.

Le registre A

Répondant plus communément au doux nom d'accumulateur, le registre A est le registre principal du microprocesseur ou de l'ordinateur.

C'est lui qui "accumule" le plus de tâches dans le fonctionnement intime du microprocesseur. En revanche, il est le seul à accepter tous les modes d'adressage (sauf le relatif bien entendu, qui s'adresse aux instructions de branchement).

Toutes les opérations arithmétiques, logiques et de comparaison sont effectuées par son intermédiaire. Sa contenance est de 8 litres... pardon : de 8 bits !

Le registre F

Nous l'avons vu, le registre F est le "porte-drapeaux" du microprocesseur. Nous l'examinerons en détail plus loin.

Le registre B

Ce registre serait relativement banal s'il n'était spécialement utilisé comme compteur par une instruction de boucle : DJNZ. Il peut être associé avec le registre C pour former un double registre.

Les registres C, D, E, H et L

Ce sont de simples registres 8 bits pouvant être associés par paires : BC, DE, HL. HL a toutefois une particularité : il supporte, tout comme A, bien qu'étant de 16 bits, tous les modes d'adressage, sauf le relatif et l'indexé, et a souvent un rôle de pointeur mémoire, à tel point que le constructeur Intel le nomme "M" dans son assembleur du 8080 ou 8085 :

```
MOV     A,M
```

correspond à :

```
LD      A,(HL)
```

et signifie : déplacer (MOVE = déplacement) l'octet mémoire pointé par M (double registre HL) dans le registre A.

Le registre SP

Nous l'avons vu, c'est un registre de 16 bits utilisé comme pointeur de la pile.

Les registres IX et IY

Ce sont également des registres 16 bits à usage général, mais restreint, car spécialisés dans le mode d'adressage indexé.

Le registre I

Registre 8 bits utilisé dans un mode de fonctionnement particulier du Z80 lié aux interruptions. Il contiendra la partie "forte" d'une adresse vers laquelle devra aller se brancher le programme en cas d'interruption émise par un organe périphérique. Il sera à la charge de ce dernier de fournir la partie "faible" de cette adresse.

Le registre R

Comme le précédent, il est d'un usage très spécial et très technique. En principe, il n'est jamais utilisé dans la programmation et contient une adresse évolutive destinée à assurer le "rafraichissement" dynamique des blocs mémoire.

Nous ne nous étendrons pas sur les registres du second bloc (A', F', B', C', D', E', H' et L') qui, après permutation avec le bloc principal se comportent exactement de la même façon que ceux de ce dernier.

Il y a un registre dont nous n'avons pas parlé jusqu'à maintenant, car n'étant pas accessible directement par programme. Pourtant il existe bel et bien et a même un rôle primordial dans le déroulement du programme.

C'est le registre 16 bits appelé PC, le compteur programme. Il contient l'adresse instantanée de l'instruction en cours d'exécution.

Lors d'une instruction JP, par exemple, le microprocesseur écrit dans le registre PC l'adresse donnée en opérande de cette instruction, provoquant la poursuite du programme à ce nouvel emplacement mémoire.

ROLE DES INDICATEURS

Les indicateurs ou "drapeaux" sont regroupés, nous l'avons vu, dans le registre F (flags) qui a la structure suivante :

S	Z	//////	H	//////	P/V	N	C
7	6	5	4	3	2	1	0

LE BIT 0 :

Indicateur ***CARRY*** (report, retenue). Il est positionné par les opérations arithmétiques, de décalage et de comparaison, ainsi que par deux instructions spéciales permettant de forcer sa valeur à 1 et de la complémenter.

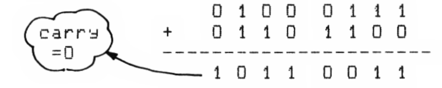
Les instructions logiques (AND, OR, XOR) le positionnent toujours à zéro.

Nous examinerons successivement quatre cas d'utilisation du bit C : l'addition, la soustraction, la comparaison et le décalage.

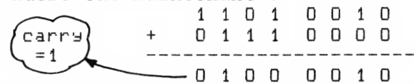
L'addition

Dans le cas de cette opération, le bit CARRY prend la valeur de la retenue s'échappant à gauche du résultat (poids fort).

Exemple d'addition 8 bits :

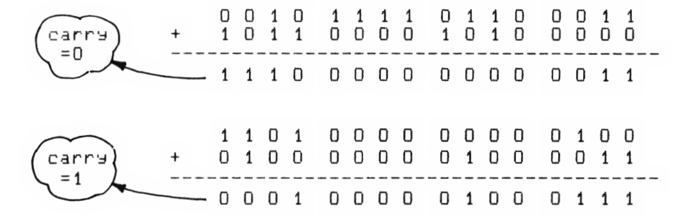


Un autre cas maintenant :



faisant apparaître une retenue, le résultat réel étant 142H.

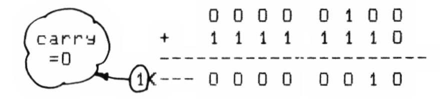
Exemple d'addition 16 bits :



La soustraction

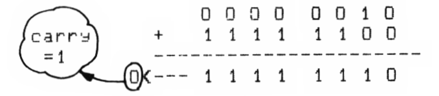
La signification du bit C est exactement la même que pour l'addition, sachant que la soustraction n'est qu'une *addition en complément à deux* (voir annexes), et que dans ce cas, le sens de la retenue doit être inversé.

Soit à effectuer 4 - 2 en soustraction 8 bits. Le complément à deux de -2 est OFEH, donc cela revient à faire 4 + OFEH :



Nous trouvons 2 comme résultat, ce qui est correct.

Effectuons maintenant 2 - 4. Le complément à deux de -4 est OFCH :



Nous voyons tout de suite que le résultat est un complément à deux pouvant s'écrire : - 2, et qui est bien celui attendu.

Comparaison

Lorsque le microprocesseur compare deux valeurs, il exécute en fait une soustraction invisible entre elles, ce qui conduit à l'interprétation suivante :

```
premiere valeur > ou = ā deuxieme valeur:  CARRY = 0
premiere valeur <      ā deuxieme valeur:  CARRY = 1
```

Lors des instructions de comparaison, la première valeur est toujours rangée dans l'accumulateur et la seconde valeur est définie en opérande par l'instruction, sur 8 bits.

Note importante : *Les valeurs comparées sont toujours considérées comme des valeurs absolues, c'est-à-dire non signées* (FOH doit être considéré comme étant supérieur à 2, par exemple).

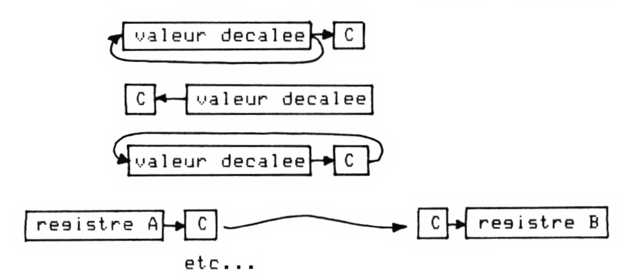
(A)	(OPERANDE)	(CARRY)
5	4	0
5	7	1
5	5	0
5	-2 (=FE)	1
-1 (=FF)	5	0

Lorsque les valeurs négatives sont remplacées par leurs compléments à deux, l'interprétation de Carry devient, en effet, plus claire.

Décalage

Dans ce mode d'utilisation, Carry peut être considéré comme un bit d'extension de la valeur décalée, un neuvième bit, en quelque sorte.

Selon le type d'instruction (décalage gauche, droit, ouvert, fermé), il participe ou non au décalage et est souvent utilisé pour véhiculer une portion d'information dans une autre.



Les instructions de décalage seront examinées en détail au chapitre suivant.

LE BIT 1 :

Indicateur ***N*** (négatif). Ce bit est positionné à 1 dans le cas où l'opération précédente était une soustraction (ou un décrément). Son utilisation est assez réduite.

LE BIT 2 :

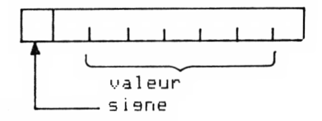
Indicateur ***P/V*** (parité/overflow = débordement). Ce bit a un double rôle dépendant du type d'instruction ayant provoqué son positionnement : il donne la parité avec les instructions logiques (nombre de bits à 1 en nombre pair ou impair) et signale un débordement pendant l'exécution des instructions arithmétiques.

Après une opération logique :

```
si A = |0 1 1 0 1 0 1 1|   , P = 0 (impair)
si A = |0 1 1 0 1 0 1 0|   , P = 1 (pair)
```

Après une opération arithmétique, l'indicateur V est positionné à 1 dans le cas où le résultat de l'opération est incohérent.

Il faut signaler ici que l'arithmétique des ordinateurs considère les valeurs 8 bits comme des valeurs signées, c'est-à-dire comprises entre -128 et +127, le bit de poids fort faisant office de signe (1 = valeur négative).



Reprenons l'un des exemples précédents :

```
                0 1 0 0  0 1 1 1        =47H
 carry      +   0 1 1 0  1 1 0 0        =6CH
  =0         -----------------------
                1 0 1 1  0 0 1 1        =B3H
```

Depuis quand l'addition de deux nombres positifs donne-t-elle un résultat négatif ? C'est pourtant ce qui se passe dans cet exemple, car B3H est en fait -4DH sous sa forme complément à deux...
Cette anomalie est signalée par l'indicateur V qui passe à 1.

L'exemple suivant nous donnait :

```
                1 1 0 1  0 0 1 0        =D2H
 carry      +   0 1 1 1  0 0 0 0        =70H
  =1         -----------------------
                0 1 0 0  0 0 1 0        =42H
```

et nous disions que, logiquement, le résultat attendu était 142H. Mais réfléchissons un peu. Si l'on considère que D2H est en fait égal à -2EH, alors :

-2EH + 70H = 42H

et le résultat est bien correct. L'indicateur V passera donc à zéro.

Pas d'la tarte hein, ces indicateurs... ?

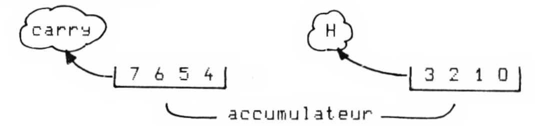
On peut résumer la situation en disant que l'indicateur V passe à 1 lorsqu'une situation paradoxale est intervenue au niveau du signe du résultat.

LE BIT 4 :

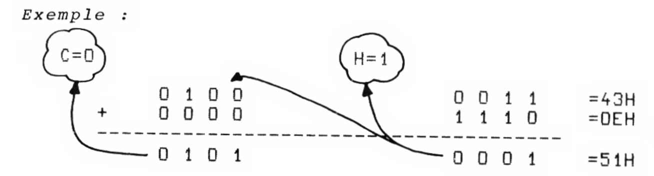
Indicateur ***H*** (Half-carry = demi-carry). Rassurez-vous tout de suite, vous n'aurez pas à utiliser très souvent cet indicateur !

Il est réservé à l'exécution d'une instruction très spéciale (DAA) ayant pour fonction de transformer une valeur binaire 8 bits en son équivalent codé en BCD (binaire codé décimal. Voir annexe).

C'est une "moitié" de Carry... d'où son nom. Il donne en effet les mêmes indications que ce dernier, mais sur une valeur de quatre bits seulement, et représente la retenue s'échappant, non du bit 7, mais du bit 3 de l'accumulateur, étant considéré que celui-ci est formé de 2 "quartets" de quatre bits :

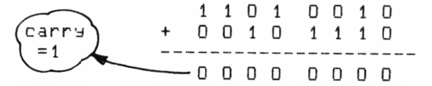


Exemple :



LE BIT 6 :

Indicateur ***Z*** (zéro). Il est positionné à 1 par certaines instructions produisant un résultat nul. Par exemple l'addition :



positionnera le bit Z à 1, indiquant par là que le résultat de l'opération est égal à zéro.

LE BIT 7 :

Indicateur ***S*** (signe). Ce drapeau est monté par certaines instructions et indique le signe de la valeur testée ou générée, ainsi que nous l'avons vu précédemment :

S = 1 ; valeur negative
S = 0 ; valeur positive (0 a 127)

et correspond au bit de poids fort de cette valeur.

Vous devez en connaître assez maintenant pour aborder en détail un jeu d'instruction type, celui du Z80 qui est, à ce jour, le plus complet. A L'ABORDAAAAAGE..... !
(N'oubliez pas de hisser tous les drapeaux !).

EXERCICES PRATIQUES DU CHAPITRE II

Nous voici arrivés à la fin du chapitre II, et vous devez en savoir assez maintenant pour pouvoir résoudre les petits exercices proposés ici.

Ne cherchez pas trop de complications, ils sont très simples, et même si vous n'en venez pas encore à bout, ce n'est pas grave. Le principal c'est d'essayer et de pratiquer ! C'est la clé de la réussite.

Une annexe donne la solution de ces exercices, mais ne la consultez qu'en dernier ressort !

EXERCICE 2.1 *: Additionner deux nombres compris entre 1 et 127, disponibles dans les registres B et C. Le résultat devra être fourni dans le registre B. Bien entendu, l'instruction d'addition ne devra pas être employée !*

EXERCICE 2.2 *: Soit le programme suivant :*

```
        LD      BC,0ABCH
        PUSH    BC
        CALL    ROUT
        JR      $                   ; arrêt
ROUT    POP     HL
        INC     HL
        INC     HL
        PUSH    HL
        RET
```

Que fait-il d'après vous ? Pouvez-vous en expliquer le mécanisme ?

EXERCICE 2.3 *: Et que pensez-vous de celui-ci ?*

```
        LD      A,3DH
        LD      (ADR),A
ADR     INC     A
        INC     A
```

Que contiendra le registre A à la fin du programme ?

EXERCICE 2.4 *: Ecrire un programme -le plus court possible- qui place la valeur 0 (zéro) dans la plus grande partie de la mémoire (considérée comme vive).*

EXERCICE 2.5 *: Une zone mémoire pointée par l'étiquette BUFFER contient une suite de dix informations, par exemple les nombres 0 à 9 sous forme ASCII, en ordre ascendant. Ecrire un programme qui inverse l'ordre de ces informations (ordre descendant).*

CHAPITRE 3

INSTRUCTIONS D'UN ASSEMBLEUR TYPE : LE Z80

Le jeu d'instructions du Z80 produit, sauf erreur, 696 codes opérations différents...(!), lesquels, après classification, deviennent fort heureusement moins rébarbatifs !

Dans les pages qui suivent, nous allons tenter un "découpage" de ce jeu d'instructions en neuf "familles", en faisant apparaître dans chacune d'elles les différents modes d'adressage qui sont permis, ceci afin d'en faciliter l'étude.

Les informations telles que : nombre de cycles horloge, temps d'exécution, code objet produit, ont été volontairement ignorés afin de ne pas surcharger les tableaux. D'ailleurs, ces informations sont rarement utilisées par celui qui programme en assembleur. Le code objet figure toutefois en annexe.

Comme toute classification, la présente possède des imperfections, et nous avons parfois eu du mal à insérer certaines instructions dans une famille plutôt que dans une autre. Des renvois ont été prévus à cet effet.

Les neuf familles (c'est un nouveau jeu aussi amusant que celui des sept...) sont celles qui ont déjà été évoquées dans le chapitre précédent, et que nous rappelons ici :

1. appels/rangements mémoire
2. chargements et échanges registres
3. fonctions arithmétiques
4. fonctions logiques auxquelles ont été ajoutées les comparaisons
5. usage général et contrôle interne

6. ruptures de séquence (branchements)
7. entrées/sorties
8. spéciales sur bits et chaînes
9. manipulations de pile.

Fallait-il inclure les instructions de pile dans la première catégorie (la pile est en mémoire elle aussi !) ? Nous avons préféré utiliserune famille séparée pour décrire une fonctionnalité qui, rappelons-le, n'existe pas sur tous les ordinateurs.

Avant d'aborder cette étude, définissons préalablement les symboles qui vont y être utilisés :

- ***ADR*** désignera toujours une valeur d'adresse de 16 bits.
- ***VAL8*** représentera une valeur de 8 bits.
- ***VAL16*** sera une valeur de 16 bits ou une adresse.
- ***DEPL*** sera spécialement utilisé dans le cas des instructions en mode relatif, et désignera un label d'adresse. Nous avons, en effet, préféré utiliser un symbole différent de ADR, afin de montrer que le code véritablement produit par l'assembleur est un déplacement relatif défini sur un octet.
- ***d*** désignera une valeur signée (+127 à -128) utilisée dans les instructions indexées.

FAMILLE 1 : APPELS/RANGEMENTS MEMOIRE

Ces fonctions sont réalisées par ***l'instruction LD.***

L'échange d'informations s'effectue de mémoire à registre, ou vice-versa.

Le pointeur mémoire peut être une adresse directe ou un double registre.

adressage	instructions		indicateurs
direct 8 bits	LD A,(ADR)	LD (ADR),A	non-affectés

La première instruction charge le registre A avec l'octet mémoire situé à l'adresse définie par ADR, la seconde range le registre A à cette adresse. Que dire d'autre sur ces instructions déjà bien connues ?

adressage	instructions		indicateurs
indirect par HL 8 bits	LD A,(HL)	LD (HL),A	non-affectés
	LD B,(HL)	LD (HL),B	
	LD C,(HL)	LD (HL),C	
	LD D,(HL)	LD (HL),D	
	LD E,(HL)	LD (HL),E	
	LD H,(HL)	LD (HL),H	
	LD L,(HL)	LD (HL),L	

LD A,(HL) provoque le chargement dans le registre A, de l'octet mémoire dont l'adresse est pointée par le double registre HL. LD (HL),A range le contenu du registre A à cette adresse. Les autres instructions agissent de même, mais pour les registres B à L.

adressage	instructions		indicateurs
indirect BC et DE 8 bits	LD A,(BC)	LD (BC),A	non-affectés
	LD A,(DE)	LD (DE),A	

Seul le registre A possède le privilège d'accéder à la mémoire par un pointeur registre BC ou DE.

adressage	instructions		indicateurs
indirect indexé 8 bits	LD (IX+d),A	LD A,(IX+d)	non-affectés
	LD (IY+d),A	LD A,(IY+d)	
	LD (IX+d),B	LD B,(IX+d)	
	LD (IY+d),B	LD B,(IY+d)	
	LD (IX+d),C	LD C,(IX+d)	
	LD (IY+d),C	LD C,(IY+d)	
	LD (IX+d),D	LD D,(IX+d)	
	LD (IY+d),D	LD D,(IY+d)	
	LD (IX+d),E	LD E,(IX+d)	
	LD (IY+d),E	LD E,(IY+d)	
	LD (IX+d),H	LD H,(IX+d)	
	LD (IY+d),H	LD H,(IY+d)	
	LD (IX+d),L	LD L,(IX+d)	
	LD (IY+d),L	LD L,(IY+d)	

Avec ces instructions, le registre considéré (A à L), est rangé à (ou chargé par) l'adresse pointée par le registre d'index (IX ou IY) augmentée de la valeur de déplacement "d".

adressage	instructions	indicateurs
immediat 8 bits indirect et indirect indexé	LD (HL),VAL8	non-affectés
	LD (IX+d),VAL8	
	LD (IY+d),VAL8	

La valeur immédiate VAL8 est chargée à l'adresse mémoire contenue dans HL ou dans un registre d'index. Dans ce dernier cas, il faut lui ajouter le déplacement relatif "d" et le mode d'adressage est alors immédiat, indirect et indexé !
Patience ! Nous verrons des exemples dans quelques instants !
Bien entendu, dans ce mode d'adressage, il ne peut y avoir que des rangements (ou écritures) mémoire.

adressage	instructions		indicateurs
direct 16 bits	LD (ADR),BC	LD BC,(ADR)	non-affectés
	LD (ADR),DE	LD DE,(ADR)	
	LD (ADR),HL	LD HL,(ADR)	
	LD (ADR),SP	LD SP,(ADR)	
	LD (ADR),IX	LD IX,(ADR)	
	LD (ADR),IY	LD IY,(ADR)	

Le contenu du double registre (16 bits) spécifié est directement rangé à l'adresse ADR et ADR+1 et vice-versa. Dans quel ordre ? Rappelons-le : registre faible (C) à l'adresse faible (ADR) et registre fort (B) à l'adresse forte (ADR+1).

adressage	instructions	indicateurs
indirect de memoire à memoire	LDI LDD	.S,Z,C inchangés. .N et H =0. .P/V =0 si BC =0, sinon=1

Voici deux instructions très puissantes dans une bcucle contrôlée. L'octet pointé par HL est chargé à l'adresse pointée par DE. De plus, BC est décrémenté de 1 et provoque le passage à zéro de l'indictateur P/V lorsqu'il atteint une valeur nulle. Mais ce n'est pas tout : les deux registres pointeurs HL et DE sont incrémentés dans le cas de l'instruction LDI (I comme incrément), et décrémentés dans le cas de l'instruction LDD (D comme décrément).

adressage	instructions	indicateurs
	Voir aussi les instructions speciales sur chaines...	

FAMILLE 2 : CHARGEMENTS REGISTRES ET ECHANGES

Cette nouvelle famille est très proche de la première, à tel point qu'elle utilise la même mnémonique du code opération "LD".

CHARGEMENTS

adressage	instructions	indicateurs
immediat 8 bits	LD A,VAL8	non-affectés
	LD B,VAL8	
	LD C,VAL8	
	LD D,VAL8	
	LD E,VAL8	
	LD H,VAL8	
	LD L,VAL8	

La valeur 8 bits représentée par VAL8 est chargée immédiatement dans le registre considéré.

Exemple : LD H,0FCH

charge la valeur hexadécimale OFC dans le registre H.

adressage	instructions	indicateurs
immediat 16 bits	LD BC,VAL16 LD DE,VAL16 LD HL,VAL16 LD SP,VAL16 LD IX,VAL16 LD IY,VAL16	non-affectés

Avec ces instructions, la paire de registres spécifiée en premier argument de l'opérande est chargée par la valeur de 16 bits VAL16.

Exemple :

```
LD      IX,4000H
LD      IY,65535
```

Les valeurs 4000 et FFFF (65535) seront respectivement chargées dans les registres d'index IX et IY.

adressage	instructions	indicateurs
registre 8 bits	LD R1,R2 R1 et R2 etant l'un des registres 8 bits: A, B, C, D, E, H, L	non-affectés

Ici, pas de restrictions, tous les coups sont permis !

Exemple :

```
LD      A,A     ; c'est ridicule...
LD      H,C     ; c'est deja mieux...
```

adressage	instructions	indicateurs
registres speciaux 8 bits	LD A,I LD A,R	H et N = 0 C inchangé S et Z selon resultat P/V = IFF
	LD I,A LD R,A	non-affectés

Ces deux types d'instructions utilisent les registres spéciaux du Z80 : I (interruption) et R (rafraichissement mémoire). Les deux instructions de chargement du registre A positionnent les indicateurs de signe (S) et de zéro (Z) selon la valeur chargée dans A. De plus, le drapeau P/V contient, après exécution, l'état d'une bascule appelée IFF, indiquant l'état des interruptions (masquées ou démasquées).

adressage	instructions	indicateurs
registre 16 bits	LD SP,HL LD SP,IX LD SP,IY	non-affectés

Le registre pointeur de pile a le privilège de pouvoir être chargé par l'un des doubles registres HL, IX ou IY.

ECHANGES

Il est temps de changer un peu de mnémonique... Le LD commençait à devenir lassant !

adressage	instructions	indicateurs
registres 16 bits (inter-blocs)	EXX	non-affectés
	EX AF,AF'	prennent l'etat du registre F'

L'instruction EXX réalise l'échange des registres BC, DE, HL avec leurs homologues BC', DE' et HL' du second bloc de registres.
La seconde instruction fait de même avec les doubles registres AF et AF'.
L'exécution de ces instructions implique l'utilisation d'un registre que le microprocesseur garde en secret et qui permet le transfert :

```
                    AF ---> registre intermediaire
                    AF'---> AF
registre intermediaire ---> AF'
```

dans le cas de l'instruction EX AF,AF', par exemple.

adressage	instructions	indicateurs
registre 16 bits	EX DE,HL	non-affectés

Cette instruction effectue l'échange des registres DE et HL du bloc primaire.

adressage	instructions	indicateurs
indirect SP 16 bits	EX (SP),HL EX (SP),IX EX (SP),IY	non-affectés

Certains diront que ces instructions sont mal placées...
Dans le cas présent, l'échange est en effet effectué entre l'un des doubles registres HL, IX et IY d'une part, et la mémoire pointée sur le registre SP, d'autre part (la pile).

Le temps est peut-être venu de se relaxer un brin... avec quelques exemples !

```
LD      IX,4000H
LD      BC,4243H        ; caracteres B et C
LD      (IX),B
LD      (IX+1),C
LD      (IX-1),41H
```

Après exécution de ce petit programme, la mémoire située aux adresses 3FFFH, 4000H et 4001H contiendra les valeurs respectives 41H, 42H et 43H, soit les caractères "A", "B" et "C" en ASCII.

```
LD      HL,4000H
LD      A,0AAH
LD      (HL),A
EX      DE,HL
LD      (4000H+1),DE
```

Maintenant, la zone mémoire 4000H à 4002H contient les valeurs hexadécimales 0AAH, OOH et 40H. OK ?

FAMILLE 3 : FONCTIONS ARITHMETIQUES

Nous passons maintenant à une famille très importante, mais un peu courte.
Il faudra se rappeler que ces instructions opèrent toujours sur des valeurs de 7 ou de 15 bits plus signe, et que les indicateurs les interprètent comme telles.

ADDITIONS avec et sans "Carry"

Ces instructions étant très proches, elles ont été regroupées dans un même tableau.

Les additions avec Carry (bit indicateur C) ***ADC*** fonctionnent comme les additions simples ***ADD***, mais ajoutent au résultat la valeur du bit C (état précédant l'instruction).

Le bit N est toujours forcé à zéro. Les autres indicateurs sont positionnés en fonction du résultat.

adressage	instructions		indicateurs
immediat 8 bits	ADD A,VAL8	ADC A,VAL8	modifiés

La valeur 8 bits est ajoutée à celle contenue dans l'accumulateur A, qui contient également le résultat. Le bit C est ajouté dans le cas de l'instruction ADC.

adressage	instructions		indicateurs
registre 8 bits	ADD A,A	ADC A,A	modifiés
	ADD A,B	ADC A,B	
	ADD A,C	ADC A,C	
	ADD A,D	ADC A,D	
	ADD A,E	ADC A,E	
	ADD A,H	ADC A,H	
	ADD A,L	ADC A,L	

Dans le cas de ces instructions, le contenu du registre considéré est ajouté à celui de l'accumulateur. L'instruction ADC opère, de plus, une addition de la valeur du bit carry avec le résultat.

adressage	instructions		indicateurs
indirect et indirect indexé 8 bits	ADD A,(HL)	ADC A,(HL)	modifiés
	ADD A,(IX+d)	ADC A,(IX+d)	
	ADD A,(IY+d)	ADC A,(IY+d)	

Là ce n'est plus le registre, mais l'octet mémoire pointé par le deuxième argument d'opérande, qui est ajouté au contenu de l'accumulateur.

adressage	instructions		indicateurs
registre 16 bits	ADD HL,BC	ADC HL,BC	modifiés
	ADD HL,DE	ADC HL,DE	
	ADD HL,HL	ADC HL,HL	
	ADD HL,SP	ADC HL,SP	

Ces instructions opèrent sur des valeurs de 16 bits (15 bits + signe). HL joue ici le rôle de double accumulateur et reçoit le résultat de l'addition de son contenu avec l'un des registres BC, DE, HL ou SP.

adressage	instructions	indicateurs
registre 16 bits	ADD IX,BC ADD IX,DE ADD IX,IX ADD IX,SP ADD IY,BC ADD IY,DE ADD IY,IY ADD IY,SP	modifiés

Dans le cas présent, l'instruction ADC manque à l'appel. Mais elle prendra sa revanche plus tard.

SOUSTRACTIONS avec ou sans "Carry"

Le principe est le même que pour l'addition, aussi nous trouverons le duo de mnémoniques ***SUB*** et ***SBC***. Avec cette dernière, la valeur du bit C est retranchée du résultat. Le bit N est toujours forcé à 1 dans une soustraction.

Histoire de compliquer un peu les choses, la syntaxe de la soustraction obéit à des règles un peu farfelues. Alors que l'addition s'écrivait par exemple ADD A,B la soustraction est privée de son premier argument et s'écrit SUB B, ce qui n'est pas plus idiot que SUB A,B mais manque d'homogénéité avec les autres instructions.

Le pire, c'est que SBC continue à s'écrire SBC A,B ... là, c'est complètement idiot ! Nous verrons plus loin que ce n'est malheureusement pas le seul cas. Mais cela ne doit pas vous décourager puisque vous êtes prévenu maintenant.

adressage	instructions		indicateurs
immediat 8 bits	SUB VAL8	SBC A,VAL8	modifiés

La valeur 8 bits représentée par VAL8 est soustraite à l'accumulateur. L'instruction SBC opère, de plus, une soustraction au résultat de la valeur du bit C (valeur prise avant exécution de l'instruction).

adressage	instructions		indicateurs
registre 8 bits	SUB A	SBC A,A	modifiés
	SUB B	SBC A,B	
	SUB C	SBC A,C	
	SUB D	SBC A,D	
	SUB E	SBC A,E	
	SUB H	SBC A,H	
	SUB L	SBC A,L	

Remarquez que l'instruction : SUB A donne toujours un résultat nul. Par contre, SBC A,A peut donner 0 ou -1 (c'est-à-dire OFFH) selon la valeur du carry...

adressage	instructions				indicateurs
indirect et indirect indexé 8 bits	SUB	(HL)	SBC	A,(HL)	modifiés
	SUB	(IX+d)	SBC	A,(IX+d)	
	SUB	(IY+d)	SBC	A,(IY+d)	

L'octet mémoire pointé par l'argument indirect est soustrait au contenu de l'accumulateur (avec carry pour le cas SBC).

adressage	instructions		indicateurs
registres 16 bits	SBC	HL,BC	modifiés
	SBC	HL,DE	
	SBC	HL,HL	
	SBC	HL,SP	

Voilà la revanche dont il était question plus haut ! Ici, pas d'instructions SUB équivalentes, aussi faut-il surveiller la valeur de C avant d'exécuter une soustraction 16 bits. C'est déjà mieux que rien, car le cousin 8080 n'a même pas cette possibilité !

INCREMENTS et DECREMENTS

Les incréments et décréments ne sont pas autre chose que des additions et des soustractions avec l'unité.

Décrémenter le registre A, par exemple, c'est lui retirer une unité. Ces instructions sont très utilisées dans les programmes.

adressage	instructions				indicateurs
registre 8 bits	INC	A	DEC	A	C inchangé, les autres modifiés
	INC	B	DEC	B	
	INC	C	DEC	C	
	INC	D	DEC	D	
	INC	E	DEC	E	
	INC	H	DEC	H	
	INC	L	DEC	L	

Prenons un exemple :

```
LD      A,40H
DEC     A
```

le registre A contiendra la valeur 3FH après exécution

adressage	instructions		indicateurs
indirect et indirect indexe	INC (HL) INC (IX+d) INC (IY+d)	DEC (HL) DEC (IX+d) DEC (IY+d)	C inchangé les autres modifiés

Dans ce cas, c'est la valeur mémoire pointée par l'opérande qui sera incrémentée ou décrémentée d'une unité.

adressage	instructions		indicateurs
registres 16 bits	INC BC INC DE INC HL INC SP INC IX INC IY	DEC BC DEC DE DEC HL DEC SP DEC IX DEC IY	non-modifiés

Ces instructions ont une fâcheuse tendance : elles ne positionnent pas les indicateurs. C'est parfois gênant, mais moins lorsqu'on le sait.

```
LD      HL,0FFFFH
INC     HL
```

Que se passe t-il d'après vous ?

adressage	instructions	indicateurs
implicite	DAA	N inchangé, les autres modifiés

Adressage implicite ? Vasisdas ?
Cela veut tout simplement dire que cette instruction, de même que quelques autres du même style, sait implicitement qu'elle doit "travailler" sur le registre A. Mais que fait-elle ?
Elle exécute ce que l'on a coutume d'appeler : un ajustement décimal. Vous allez tout comprendre.

On vient d'effectuer l'addition de deux nombres : 8 et 6 :

```
LD      A,8
LD      B,6
ADD     A,B
```

Le registre A contient maintenant la valeur binaire 0000 1110 soit OEH en hexadécimal. Mais si nous voulons obtenir le résultat en décimal, il y a une solution : l'instruction DAA. Effectuée à la suite de l'addition, elle produira le résultat 0001 0100 dans l'accumulateur, c'est-à-dire 14... en hexadécimal. Or, 8 et 6 font bien 14 ! Cela revient à faire une addition décimale. Sur des grands nombres, il faut utiliser Carry et "passer la balle" à l'octet suivant.

L'instruction DAA travaille par quartet (4 bits = 1/2 octet). Lorsque la valeur du quartet droit de A dépasse 9 ou que le bit H est à 1, l'accumulateur est incrémenté de 6. Si maintenant la valeur du quartet gauche est supérieure à 9 ou que le bit C est à 1, ce quartet est incrémenté de 6. C'est une manière de transformer le binaire en BCD (binaire code décimal).

FAMILLE 4 : LES FONCTIONS LOGIQUES

Les instructions logiques comprennent les intersections (*AND*), les réunions (*OR*), les disjonctions (*XOR*), les complémentations, les comparaisons (puisque ce sont des comparaisons logiques et non arithmétiques) et les décalages.

INTERSECTIONS

Ces instructions, comme la plupart de celles qui suivent, adressant implicitement le registre A, celui-ci ne figure pas dans l'opérande (voir le petit accrochage au sujet de SUB/SBC...).

adressage	instructions	indicateurs
immediat 8 bits	AND VAL8	C et N = 0 P = parité H = 1 S et Z modifiés

Une intersection logique est effectuée entre la valeur 8 bits VAL8 et l'accumulateur. Le résultat est dans A.
On peut se demander pourquoi l'indicateur H est mis à 1 ?
Notez le bit P qui indique la parité du résultat.
S et Z indiquent respectivement si la valeur est négative ou nulle, lorsqu'ils ont la valeur 1.

adressage	instructions	indicateurs
indirect et indirect indexé 8 bits	AND (HL) AND (IX+d) AND (IY+d)	voir AND immediat

La valeur pointée par l'opérande est placée en intersection avec A. Résultat dans A.

```
LD      A,0AAH
LD      HL,4000H
LD      (HL),A
INC     A                 ; A = 0ABH maintenant
AND     (HL)
```

L'accumulateur contiendra la valeur OAAH.

adressage	instructions	indicateurs
registre 8 bits	AND A AND B AND C AND D AND E AND H AND L	voir AND immediat

Une intersection est effectuée entre le registre donné en opérande et l'accumulateur.
Mais a quoi peut donc bien servir : AND A ? A priori, à rien ! Elle est toutefois à inscrire à la liste des "trucs" du programmeur. Nous allons prendre un exemple : Vous vous souvenez que l'instruction LD ne positionne pas les indicateurs. Alors, comment savoir si la valeur lue dans A est nulle, par exemple ? Il suffira de faire :

```
LD      A,(ADR)
AND     A
```

et les deux indicateurs S et Z seront positionnés, renseignant ainsi sur la nature de la valeur chargée.

REUNIONS

adressage	instructions	indicateurs
immediat 8 bits	OR VAL8	C et N = 0 P = parité H = 1 S et Z modifiés

La valeur immédiate VAL8 est réunie logiquement avec le contenu du registre A.

```
LD      A,8
OR      4
```

Le registre A contient maintenant la valeur 12 (OCH).

adressage	instructions	indicateurs
indirect et indirect indexé 8 bits	OR (HL) OR (IX+d) OR (IY+d)	voir OR immediat

Il n'y a vraiment rien à dire, alors un petit exemple :

```
LD      IX,TRUC
LD      BC,1234H
LD      (IX),B
LD      (IX+1),C
AND     0                 ; oui: AND z.e.r.o!
OR      (IX)
OR      (IX+1)
```

Le registre A contiendra la valeur 36H, et en verra 36 chandelles du même coup !

adressage	instructions	indicateurs
registre 8 bits	OR A OR B OR C OR D OR E OR H OR L	voir OR immediat

OR A possède, tout comme AND A, le pouvoir de faire "re-sortir" les indicateurs Z et S d'une valeur chargée dans l'accumulateur. Un autre exemple ? Tenez :

```
LD      BC,0AFEH-2
LD      A,0
OR      B
OR      C
```

Le registre A a toutes les chances de contenir la valeur OFEH !

DISJONCTIONS

La disjonction, appelée aussi OU exclusif, est réalisée par les instructions suivantes :

adressage	instructions	indicateurs
immediat 8 bits	XOR VAL8	C et N = 0 P = parité H = 1 S et Z modifiés

```
LD      A,0AFH
XOR     5
```

et le registre A contiendra la valeur OAAH. Débile, non ?

adressage	instructions	indicateurs
indirect et indirect indexé 8 bits	XOR (HL) XOR (IX+d) XOR (IY+d)	voir XOR immediat

Voici un exemple : On veut obtenir alternativement les valeurs 0, 1, 0, 1....dans un octet mémoire, à chaque fois que l'on passe à un endroit précis du programme. Ce programme aura la structure :

```
        LD      HL,BASC         ;
        LD      (HL),0          ; init octet bascule
ETIQ    ...
        ...
        LD      A,(HL)
        XOR     1               ; changement d'etat
        LD      (HL),A          ; rangement bascule
        ...
        ...
        JR      ETIQ            ; suite traitement
```

adressage	instructions	indicateurs
registre 8 bits	XOR A XOR B XOR C XOR D XOR E XOR H XOR L	voir XOR immediat

Encore une astuce de programmeur : le XOR A, bien que paraissant complètement ignare, est en fait génial (c'est toujours comme çà...), car il permet, par exemple, de mettre l'accumulateur à zéro avec un simple octet (LD A,0 en occupe deux), et quelle que soit sa valeur précédente.

```
XOR     A
LD      B,A
LD      C,A
```

Les registres A, B et C seront remis à zéro.

COMPLEMENTATION et NEGATION

Les deux instructions qui suivent réalisent respectivement le complément à 1 (NOT) ou complémentation, et le complément à 2 ou négation de l'accumulateur.

adressage	instructions	indicateurs
implicite	CPL	N et H = 1 les autres inchangés
	NEG	N = 1 les autres modifiés

Exemple :

```
LD      A,14H
CPL
NEG
```

après l'instruction CPL, l'accumulateur contiendra OEBH, et après le NEG il contiendra 15H.

```
0 0 0 1  0 1 0 0  --->  A au depart
1 1 1 0  1 0 1 1  --->  complement ā 1 de A
0 0 0 1  0 1 0 0  --->  nouveau complement ā 1
               1  --->  +1 donne:
0 0 0 1  0 1 0 1  --->  le complement a 2
```

COMPARAISONS

Ce type d'instruction est très utilisé également dans les programmes. Pour interpréter correctement le bit Carry, les deux valeurs comparées, dont l'une est toujours dans l'accumulateur, doivent être considérées comme non-signées si l'on ne veut pas se retrouver aux urgences à l'hôpital psychiatrique le plus proche (le psychiatre, lui, ne tarderait pas à faire certaines... comparaisons !).

Suivez-moi bien, Docteur, la comparaison de deux valeurs n'est en fait qu'une soustraction interne, provoquant la montée du bit Carry s'il n'y a pas de report du bit 7 dans l'opérande, indiquant par là que ce dernier est supérieur à l'accumulateur (c'est très clair), à moins que les deux quantités diffèrent en signe, auquel cas le sens de Carry doit être inversé...

Pourquoi se creuser les méninges, alors qu'il suffit tout simplement de consulter le tableau suivant :

Carry	A et operande en valeurs absolues
0	A > ou = operande
1	A < ā operande

D'autre part, Z indique toujours une égalité s'il prend la valeur 1. Les autres indicateurs retrouvent leurs significations habituelles.

adressage	instructions	indicateurs
immediat 8 bits	CP VAL8	N = 1 les autres modifiés

La valeur immédiate VAL8 est comparée avec l'accumulateur.

Exemples :

```
LD      A,5
CP      4       ; C = 0, A > 4
CP      7       ; C = 1, A < 7
CP      5       ; C = 0, A = 5
CP      0FEH    ; C = 1, A < 0FEH
LD      A,0FFH
CP      5       ; C = 0, A > 5
```

adressage	instructions	indicateurs
registre 8 bits	CP A CP B CP C CP D CP E CP H CP L	N = 1 les autres modifies

Le registre considéré est comparé avec l'accumulateur. Le cas de l'instruction CP A est toutefois énigmatique, il faut l'admettre ! Elle peut néanmoins être utilisée pour mettre C à zéro, sans modifier les autres indicateurs (V et H par exemple).

adressage	instructions	indicateurs
indirect et indirect indexé 8 bits	CP (HL) CP (IX+d) CP (IY+d)	N = 1 les autres modifiés

Pas de difficulté ici...

adressage	instructions	indicateurs
indirect avec increment/ decrement	CPI CPD	N = 1 C inchangé P/V = 1 sauf si BC = 0 Z = 1 si A = (HL)

Nous retrouvons ici deux instructions équivalentes à celles rencontrées dans la première famille : LDI et LDD.
L'octet pointé par HL est comparé à l'accumulateur, et s'il y a égalité, l'indicateur Z est mis à 1. Dans le même temps, le double registre BC est décrémenté, et P/V passe à zéro lorsque celui-ci atteint une valeur nulle.

Enfin, et selon l'instruction, HL est incrémenté pour CPI et décrémenté pour CPD.

adressage	instructions	indicateurs
	voir aussi instructions speciales sur chaines	

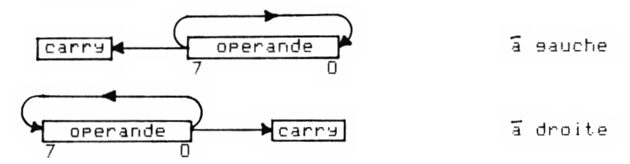
DECALAGES

Globalement, il y a cinq grands types de décalages :

- circulaires à gauche ou à droite,
- circulaires à travers carry, gauche/droite,
- ouverts arithmétiques, gauche/droite,
- ouverts logiques, à droite (seulement),
- circulaires BCD, gauche/droite.

Les cinq groupes vont maintenant être examinés, avec leurs différents modes d'adressage.

Circulaires gauche et droite



Les bits de l'opérande sont translatés vers la gauche (left) ou vers la droite (right). Le bit sortant d'un côté, entre de l'autre, tout en étant recopié dans Carry.

adressage	instructions		indicateurs
registre 8 bits	RLC A RLC B RLC C RLC D RLC E RLC H RLC L	RRC A RRC B RRC C RRC D RRC E RRC H RRC L	H = N = 0 S, Z, C et P modifiés
	RLCA	RRCA	H = N = 0 S, Z, P inchangés C modifié

RLC = Rotate Left Circular, RRC = Rotate Right Circular.

Le registre opérande est décalé circulairement d'une position binaire vers la gauche (left) ou vers la droite (right). Le bit sortant est recopié dans Carry.
Les instructions RLCA/RRCA viennent du microprocesseur 8080, et effectuent les mêmes opérations que les RLC A et RRC A, mis à part que les indicateurs S, Z et P sont positionnés par ces dernières. En contrepartie, RLCA/RRCA n'occupent qu'un octet de code contre deux pour RLC A/RRC A.

Exemple :

```
LD      B,0A4H
RLC     B
```

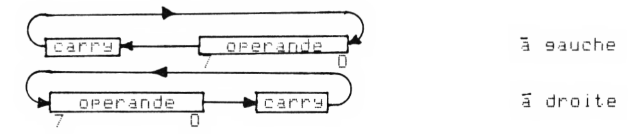
Le registre B contiendra alors la valeur 49H, et l'indicateur C sera à 1.

adressage	instructions		indicateurs
indirect et indirect indexé 8 bits	RLC (HL) RLC (IX+d) RLC (IY+d)	RRC (HL) RRC (IX+d) RRC (IY+d)	idem RCL A

L'octet mémoire pointé par l'opérande est décalé circulairement d'une position binaire vers la gauche ou vers la droite. Le bit sortant est recopié dans Carry.

Circulaires à travers Carry, gauche/droite

Imaginez que l'opérande ne fasse plus 8 bits, mais 9... et qu'il soit décalé circulairement : vous aurez une idée exacte de ce que peut être ce type de décalage.



Les 8 bits de l'opérande sont translatés vers la gauche ou vers la droite, d'une position binaire. Le bit sortant d'un côté entre dans Carry, l'ancienne valeur de Carry entre de l'autre côté de l'opérande. Tout se passe comme un décalage circulaire sur 9 bits.

adressage	instructions		indicateurs
registre 8 bits	RL A RL B RL C RL D RL E RL H RL L	RR A RR B RR C RR D RR E RR H RR L	H = N = 0 S, Z, C et P modifiés
	RLA	RRA	H = N = 0 C modifié S, Z, P inchangés

RL = Rotate Left, RR = Rotate Right.

Même remarque que ci-dessus, concernant les instructions RLA et RRA qui sont issues du microprocesseur 8080.

adressage	instructions		indicateurs
indirect et indirect indexé 8 bits	RL (HL) RL (IX+d) RL (IY+d)	RR (HL) RR (IX+d) RR (IY+d)	idem RL A

L'octet mémoire pointé par l'opérande est décalé circulairement d'une position binaire, à gauche ou à droite.

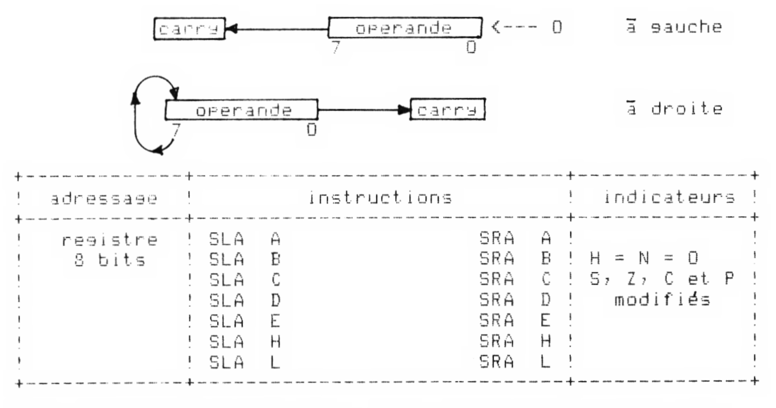
Exemple :

```
        LD      IX,4000H
        LD      A,0A4H
        OR      A               ; Carry = 0
        LD      (IX+3),A
        RL      (IX+3)
```

L'octet mémoire situé à l'adresse 4003H contiendra alors la valeur 49H et Carry sera positionné à 1.

Ouverts arithmétiques gauche ou droite

Ces décalages sont qualifiés d'**ouverts**, car le bit sortant de l'opérande à gauche ou à droite est perdu, alors que le bit entrant est à zéro, et d'**arithmétiques** parce que le décalage à droite conserve l'intégrité du signe (bit 7 de l'opérande).



adressage	instructions		indicateurs
registre 8 bits	SLA A SLA B SLA C SLA D SLA E SLA H SLA L	SRA A SRA B SRA C SRA D SRA E SRA H SRA L	H = N = 0 S, Z, C et P modifiés

SLA = Shift Left Arithmetic, SRA = Shift Right Arithmetic.
Comme tout cela semble évident, prenons un petit exemple :

```
        LD      B,0A5H
        SRA     B
```

Le registre B contiendra la valeur 4AH et l'indicateur Carry sera mis à 1.

adressage	instructions		indicateurs
indirect et indirect indexé 8 bits	SLA (HL) SLA (IX+d) SLA (IY+d)	SRA (HL) SRA (IX+d) SRA (IY+d)	idem à SLA A

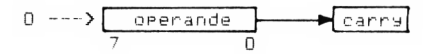
Exemple :

```
LD      IY,4000H
LD      A,0A5H
LD      (IY+3),A
SRA     (IY+3)
```

L'octet mémoire situé à l'adresse 4003H contiendra la valeur OD2H et l'indicateur Carry sera mis à 1.

Ouverts logiques à droite

Pour contrebalancer le SRA qui recopie le bit de signe après décalage, voici un nouveau type d'instruction qui n'offre pas cet inconvénient (ou cet avantage, c'est à voir...).



adressage	instructions	indicateurs
registre 8 bits	SRL A SRL B SRL C SRL D SRL E SRL H SRL L	H = N = 0 S, Z, C et P modifiés

SRL = Shift Right Logical.

Exemple :

```
LD      B,83H
SRL     B
```

Le registre B contiendra la valeur 41H, et Carry passera à 1.

adressage	instructions	indicateurs
indirect et indirect indexé 8 bits	SRL (HL) SRL (IX+d) SRL (IY+d)	idem SRL A

L'octet mémoire pointé par l'opérande sera décalé logiquement de 1 position vers la droite. Avec l'instruction SRL, on peut réaliser des décalages ouverts sur 16 bits.

Exemple :

Soit à décaler vers la droite le contenu du registre BC.

```
LD      BC,2345H
SRL     B               ; B --> carry
RR      C               ; Carry ---> C
```

Au moment du SRL, le registre B est décalé vers la droite et le bit sortant entre dans Carry. Le RR permet alors de transmettre ce bit dans le registre C qui est également décalé d'une position vers la droite.

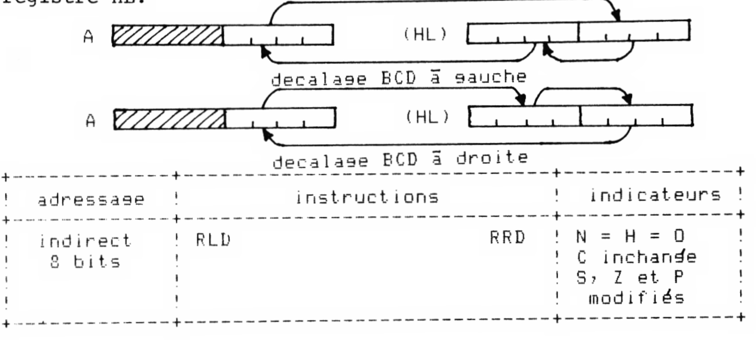
Circulaires BCD gauche et droite

Ce dernier groupe de décalages concerne deux instructions particulièrement puissantes en arithmétique décimale.

Tous les décalages étudiés jusqu'à présent se faisaient sur une longueur d'un bit, qui est la plus petite unité du système de numération binaire.

En arithmétique décimale, la plus petite unité est exprimée sur quatre bits (quartet) représentant des valeurs de zéro à neuf. Il faut donc s'attendre à ce que les décalages BCD (Binaire code decimal) s'effectuent également sur une longueur d'un quartet.

Dans le cas de ces instructions, l'accumulateur sera utilisé comme registre de manoeuvre et l'opérande sera le pointeur registre HL.



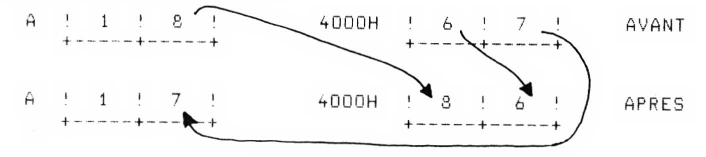
adressage	instructions		indicateurs
indirect 8 bits	RLD	RRD	N = H = 0 C inchange S, Z et P modifiés

RLD = Rotate Left Decimal, RRD = Rotate Right Decimal.

Exemple :

```
LD      B,67H
LD      HL,4000H
LD      (HL),B
LD      A,18H
RRD
```

Après exécution, l'octet mémoire 4000H contient 86H et le registre A contient 17H. Il est à noter que le quartet fort de A n'a pas été altéré.



FAMILLE 5 : USAGE GENERAL ET CONTROLES INTERNES

C'est la famille des bras-cassés, les deshérités... On ne savait pas où les mettre, alors on les a mis là.

Opérations directes sur Carry

adressage	instructions	indicateurs
implicite	SCF	C = 1 H = N = 0
	CCF	C modifié N = 0

La première instruction force Carry à prendre la valeur 1 (Set Carry Flag), alors que la seconde inverse son état (Complement Carry Flag).

Les instructions Méridionales

Dans l'équipe des concepteurs du 8080 et du Z80, il devait y avoir un Méridional au tempérament typé. Qu'il soit chaudement félicité pour avoir songé à inclure les deux instructions suivantes :

adressage	instructions	indicateurs
implicite	NOP HALT	non-affectés

La première instruction n'exécute... aucune opération (No Operation)et a deux utilisations principales. Lorsque dans un programme on doit supprimer certaines instructions, on peut les remplacer par des codes NOP au niveau du code objet, en utilisant un programme utilitaire spécial (patcheur). Cette opération évite ainsi de ré-assembler le programme, surtout s'il est très long. La deuxième utilisation est liée à la notion de temps. Lorsque le programme doit fournir des ordres d'entrées/sorties notamment, et en des temps précis, il peut être nécessaire de corriger telle ou telle partie en lui ajoutant des instructions NOP, afin de laisser s'écouler quelques microsecondes et d'arriver ainsi au synchronisme voulu. C'est ce qui s'appelle passer son temps à ne rien faire !

L'instruction HALT, par contre, n'y va pas par quatre chemins : elle arrête tout simplement le déroulement du programme. Celui-ci ne pourra être relancé que par un évènement matériel extérieur (Reset ou interruption). Mais si le déroulement du programme est stoppé, le microprocesseur continue à travailler en exécutant des NOP internes pour assurer le rafraîchissement des mémoires.

Les instructions liées aux interruptions

Le microprocesseur Z80 possède trois modes différents pour traiter les interruptions. Nous examinerons très rapidement les instructions permettant de passer d'un mode à l'autre.

Mais peut-être vous demandez-vous ce qu'est au juste une interruption ? C'est un évènement matériel extérieur provoquant l'interruption du programme en cours, le traitement spécifique à l'interruption reçue (petit programme appelé programme

d'interruption, comme il se doit), et la reprise du programme interrompu.

C'est une sorte de "cri d'alarme" venant d'un périphérique, pour dire : "J'ai fini", ou "attention, j'envoie un message", ou quelque chose de ce genre.

adressage	instructions	indicateurs
implicite	EI DI IM 0 IM 1 IM 2	non-affectés

Les deux premières instructions sont utilisées respectivement pour activer (Enable Interrupt), et désactiver (Disable Interrupt) le système de prise en compte des interruptions du microprocesseur.
Les trois instructions suivantes permettent le passage dans les différents modes d'interruption évoqués plus haut (le 8080 ne possède que le mode 0).

FAMILLE 6 : LES RUPTURES DE SEQUENCE

Ces instructions, appelées aussi "branchements", font tout l'aspect "dynamique" des programmes qui, sans elles, auraient une simple allure d'ordonnance, du style :

Faites ceci,
faites cela,
et puis encore ceci,
et après seulement, cela...

Maintenant, nous pouvons écrire :

Si... ceci, alors faites cela, sinon...

Nous pouvons distinguer 3 groupes de branchements :

- les sauts,
- les appels de sous-programmes,
- les retours de sous-programmes.

mais aussi trois modes de branchements :

- absolu,
- relatif,
- en page zéro.

LES SAUTS

Vous souvenez-vous des instructions JP ? Les voici :

adressage	instructions		indicateurs
absolu (direct)	JP ADR		non-affectés
	JP C,ADR	JP NC,ADR	
	JP Z,ADR	JP NZ,ADR	
	JP M,ADR	JP P,ADR	
	JP PE,ADR	JP PO,ADR	

La première instruction (***JP ADR***) est qualifiée de branchement inconditionnel : le saut est effectué de manière systématique à l'adresse ADR. Là il doit y avoir un programme bien sûr... sinon c'est la partie de pêche !

Les instructions suivantes sont qualifiées de conditionnelles, c'est-à-dire que le branchement ne s'effectuera que si la condition testée est satisfaite.

Bien entendu, les instructions conditionnelles ne doivent se présenter, dans le programme, qu'à la suite d'instructions ayant positionné les indicateurs.

```
....
CP      8
JP      Z,EGAL   ; branchement si A = 8
JP      C,INF    ; branchement si A < 8
....             ; A > 8
```

Dans cet exemple, l'instruction CP positionne les indicateurs Z et C. Le branchement conditionnel JP Z teste la présence du drapeau Z et provoque le branchement à l'adresse EGAL si ce dernier est à 1, indiquant par là que l'accumulateur est égal à 8. Dans le cas contraire, le programme se poursuit en séquence et "tombe" sur l'instruction JP C qui, elle, teste la présence de l'indicateur C. Si ce dernier est monté, un branchement est effectué à l'adresse INF. Si ce n'est pas le cas, nous pouvons avoir la certitude que A est supérieur à 8, puisqu'il ne satisfait pas les tests précédents, et le programme se poursuit en séquence.

adressage	instructions	indicateurs
indirect	JP (HL)	non-affectés
	JP (IX)	
	JP (IY)	

Jusqu'alors, nous avions l'habitude d'utiliser une adresse de branchement en opérande des instructions JP. A partir de maintenant, nous pourrons effectuer un branchement, non plus à une adresse donnée, mais à une adresse **calculée**, contenue dans l'un des registres HL, IX ou IY.

Exemple :

```
LD      HL,(ADR)
INC     HL
LD      A,L
AND     0FEH
LD      L,A
JP      (HL)
```

Dans cet exemple, le branchement sera effectué à une adresse a priori inconnue, mais dont on sait qu'elle sera toujours paire. Là encore, il faudra agir avec beaucoup de sagesse et se méfier des imprévus !

Mais, voici venir en sautillant... un groupe qui vous rappellera quelques souvenirs : les branchements relatifs.

adressage	instructions	indicateurs
relatif	JR DEPL JR C,DEPL JR NC,DEPL JR Z,DEPL JR NZ,DEPL	non-affectés
	DJNZ DEPL	

DEPL représente une étiquette ou label indiquant l'endroit où l'on souhaite effectuer le branchement. Le programme assembleur, qui est intelligent calculera la différence entre l'adresse de ce label et l'adresse de l'instruction de branchement, et la générera dans le code objet sous la forme d'un octet signé permettant un déplacement de 127 octets en avant et de 128 en arrière.
Les instructions JR du tableau sont en tout point identiques aux JP étudiées plus haut, sauf que le code généré est plus court et que le programme ainsi constitué peut être translaté moyennant certaines précautions.

```
        LD      B,0
BOUCL   INC     B
        JR      NZ,BOUCL
        ....
```

Le petit programme ci-dessus représente ce que l'on appelle une boucle. Tant que le registre B est différent de zéro, un branchement est effectué à l'adresse BOUCL. Mais fatalement, B atteindra la valeur OFFH et, après incrémentation de 1, il repassera à zéro. Le drapeau Z sera alors mis à 1, le branchement, cette fois, ne sera pas effectué, et le programme sortira de cette boucle infernale.

La dernière instruction : ***DJNZ***, est un peu particulière (Décrémente et "Jump" si non zéro), et utilise le registre B pour accomplir sa triste besogne... si bien que le petit programme ci-dessus pourra maintenant s'écrire :

```
        LD      B,0
BOUCL   DJNZ    BOUCL
        ....
```

Ici, on décrémente B au lieu de l'incrémenter, mais le résultat est le même. C'est plus simple, hein ?

LES APPELS DE SOUS-PROGRAMMES

Le CALL de l'assembleur est le GOSUB du Basic. Cette instruction, dont il a été question dans les premiers chapitres, effectue un branchement vers le sous-programme dont l'adresse

est spécifiée en zone opérande, non sans avoir effectué une sauvegarde dans la pile, de l'adresse de retour.

Ce dernier point différencie le CALL du simple JP, avec lequel il a des ressemblances très frappantes.

Ainsi, il y a un CALL inconditionnel et des CALL conditionnels. Par contre, il n'y a pas de CALL utilisant le mode d'adressage relatif. Et c'est gênant. De plus, il n'existe qu'un seul mode d'adressage : le direct.

adressage	instructions		indicateurs
absolu (direct)	CALL ADR		non-affectes
	CALL C,ADR	CALL NC,ADR	
	CALL Z,ADR	CALL NZ,ADR	
	CALL P,ADR	CALL M,ADR	
	CALL PO,ADR	CALL PE,ADR	

Voilà qui n'est déjà pas si mal !

Exemple :

```
LD      BC,VAL
CALL    MULT
LD      (RESULT),BC
```

On peut imaginer que ce petit programme, après avoir chargé deux valeurs dans ses registres B et C, appelle une routine qui en effectue la multiplication et fournit le résultat dans le double registre BC.
A l'exécution du CALL, l'adresse de l'instruction suivante (LD (RESUL),BC) est mise en pile, et un branchement à l'adresse MULT est effectué. En fin d'exécution de la routine, une instruction RETour provoque un dépilage, et par suite un branchement sur l'instruction LD (RESUL),BC. Génial, hein ?

Mais il peut être souhaitable de n'appeler la routine MULT que si le registre A est différent de zéro, par exemple. Dans ce cas, nous écrirons :

```
CP      0
CALL    NZ,MULT
```

et le tour sera joué !

LES RETOURS DE SOUS-PROGRAMMES

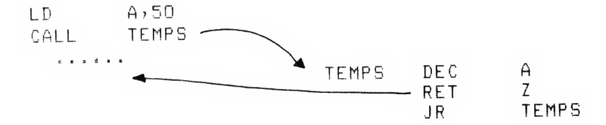
C'est bien joli d'appeler un sous-programme, mais il faut pouvoir en revenir. Les instructions suivantes sont là pour ça.

adressage	instructions		indicateurs
indirect par la pile	RET		non-affectés
	RETI		
	RETN		
	RET C	RET NC	
	RET Z	RET NZ	
	RET P	RET M	
	RET PO	RET PE	

Lorsqu'une instruction de retour est exécutée, le micro-processeur provoque un dépilage et place l'adresse extraite dans son registre PC qui est (vous en souvenez-vous ?) le pointeur d'emplacement des instructions. Le programme se poursuit donc à cette adresse.

Les trois premières instructions du tableau sont des retours inconditionnels de sous-programme. RETI et RETN sont spécialisés dans les sous-programmes d'interruption, et nous n'en parlerons pas davantage.

Les instructions suivantes sont également des retours, mais seulement s'ils obéissent à certaines conditions. Dans le cas contraire, ce sont des instructions ineffectives (il en est de même pour les branchements conditionnels en général).



Le programme ci-dessus appelle une routine déterminant un temps proportionnel à la valeur donnée dans l'accumulateur. Si nous avions écrit : LD A,100, le temps passé dans la routine aurait été pratiquement le double de celui-ci.

Nous voyons que dans la routine TEMPS, le registre A est décrémenté jusqu'à ce qu'il atteigne une valeur nulle. C'est seulement dans ce cas que le retour au programme appelant sera effectué. Nous aurions pu écrire également :

```
TEMPS   DEC    A
        JR     NZ,TEMPS
        RET
```

mais ce n'est pas drôle : vous y aviez déjà pensé !

LES APPELS DE SOUS-PROGRAMMES EN PAGE ZERO

Ce sont des CALL, mais présentant deux différences essentielles :

- Le code opération de ces instructions n'occupe qu'un octet contre trois pour les CALL classiques ;

- Ils se réfèrent à des sous-programmes d'adresses fixes, situés en début de mémoire (page zéro).

adressage	instructions	indicateurs
direct	RST 0	
	RST 8	non-affectés
	RST 10H	
	RST 18H	
	RST 20H	
	RST 28H	
	RST 30H	
	RST 38H	

Comme on peut s'en douter, les sous-programmes appelés seront implantés aux adresses 0, 8, 10H... et 38H. Dans le mode 0 du Z80 et qui est également celui du 8080, l'apparition d'une interruption déclenche un appel à l'une de ces adresses qui contiennent alors des routines d'interruption. Ces instructions portent aussi le nom de ReSTart.

Attention toutefois : ces sous-programmes sont, la plupart du temps, réservés au programme système, et implantés en mémoire morte.

FAMILLE 7 : LES ENTREES/SORTIES

Il n'existe, dans les microprocesseurs, que deux instructions permettant le dialogue avec le monde extérieur (périphériques):

- L'instruction IN pour les entrées,
- L'instruction OUT pour les sorties.

(Les termes entrée et sortie sont toujours relatifs au microprocesseur et sont, bien sûr, inversés si l'on se réfère aux périphériques).

Le processus de dialogue est le suivant : le microprocesseur envoie le contenu d'un registre vers un organe périphérique répondant à l'adresse donnée en opérande de l'instruction OUT, ou reçoit dans ce registre l'information envoyée par l'organe périphérique dont l'adresse figure en opérande de l'instruction IN.

Le registre en question est, ordinairement, l'accumulateur, mais le Z80 permet l'utilisation de tous les simples registres A à L.

En général, l'adresse du périphérique est spécifiée directement dans l'opérande de l'instruction, sur 8 bits. Là encore, le Z80 permet d'utiliser le registre C pouvant contenir une adresse calculée.

adressage	instructions		indicateurs
direct 8 bits	OUT (VAL8),A	IN A,(VAL8)	non-affectés

Le contenu du registre A est envoyé à l'organe périphérique d'adresse VAL8, dans le cas de l'instruction OUT et est chargé par l'information émise par le périphérique d'adresse VAL8, dans le cas de l'instruction IN.

adressage	instructions		indicateurs
indirect registre 8 bits	OUT (C),A	IN A,(C)	non-affectés par OUT
	OUT (C),B	IN B,(C)	
	OUT (C),C	IN C,(C)	
	OUT (C),D	IN D,(C)	pour IN:
	OUT (C),E	IN E,(C)	S, Z, H et P modifiés
	OUT (C),H	IN H,(C)	C inchangé
	OUT (C),L	IN L,(C)	N = 0

Dans le cas de l'instruction OUT, le contenu du registre (A à L) est envoyé vers l'organe périphérique dont l'adresse figure dans le registre C. On peut considérer que l'adressage du périphérique est indirect, dans ce cas (pointeur registre). Le processus est inversé pour l'instruction IN.

adressage	instructions		indicateurs
indirect registre avec increment/ decrement	INI	IND	N = 1 C inchangé Z=1 si B=0 les autres indeterminés
	OUTI	OUTD	

Ces nouveaux types d'instructions ne nous sont pas totalement inconnus. Nous les avions déjà rencontrés dans le cas des instructions LD et CP.
L'adresse du périphérique est toujours contenue dans le registre C, mais l'information émise (OUT) ou lue (IN) n'est plus dans un registre, mais en mémoire, pointée par HL. Il y a donc une double indirection, pour l'adresse et pour la donnée.
A chaque exécution de l'une de ces instructions, le registre B est décrémenté de 1, HL est incrémenté pour les instructions OUTI et INI, et décrémenté pour les instructions OUTD et IND.

adressage	instructions	indicateurs
	voir aussi les instructions speciales sur chaines	

FAMILLE 8 : INSTRUCTIONS SPECIALES SUR BITS ET CHAINES

Cette famille concerne, en effet, des instructions spéciales que nous ne rencontrons pas de manière courante sur les autres microprocesseurs, et parfois même sur des ordinateurs plus puissants.

INSTRUCTIONS SUR BITS

Ordinairement, le microprocesseur agit sur des valeurs de 8 bits, qu'elles soient dans un registre ou en mémoire, exceptionnellement sur des quartets.

Les instructions sur bits permettent d'augmenter encore le degré de finesse, en réalisant les trois fonctions suivantes :

- mise à 1 d'un bit (instruction SET),
- mise à zéro d'un bit (instruction RES),
- test de la valeur d'un bit (instruction BIT),

que le bit en question soit indifféremment dans un registre ou en mémoire.

adressage	instructions		
	test	mise à 1	mise à 0
registre 8 bits	BIT b,A	SET b,A	RES b,A
	BIT b,B	SET b,B	RES b,B
	BIT b,C	SET b,C	RES b,C
	BIT b,D	SET b,D	RES b,D
	BIT b,E	SET b,E	RES b,E
	BIT b,H	SET b,H	RES b,H
	BIT b,L	SET b,L	RES b,L
indicateurs	Z modifié N=0, H=1 C inchangé	inchangés	inchangés

Le symbole **b** représente le numéro du bit, et prend les valeurs 0 à 7.

7	6	5	4	3	2	1	0

L'instruction BIT charge dans l'indicateur Z l'inverse du bit testé : si b = 1, Z = 0 et vice-versa.

Exemple :

```
LD      A,8
SET     1,A
BIT     1,A
JR      NZ,EGAL
```

L'instruction SET forcera à 1 le bit 1 de l'accumulateur, qui contiendra alors la valeur OAH. Le test du bit 1 sera alors effectué par l'instruction BIT, et l'indicateur Z sera positionné à zéro (puisque ce bit est à 1...). Un branchement sera ensuite effectué à l'adresse EGAL.

adressage	instructions		
indirect et indirect indexé 8 bits	test	mise à 1	mise à 0
	BIT b,(HL)	SET b,(HL)	RES b,(HL)
	BIT b,(IX+d)	SET b,(IX+d)	RES b,(IX+d)
	BIT b,(IY+d)	SET b,(IY+d)	RES b,(IY+d)

indicateurs	Z modifié N=0, H=1 C inchangé	inchangés	inchangés

Même opération que pour le tableau précédent, mais le bit en question se trouve cette fois en mémoire, pointé par l'un des registres spécifié en opérande.

INSTRUCTIONS SUR CHAINES

Ce type d'instructions, à l'inverse des instructions bits qui travaillaient sur des quantités inférieures à l'octet, opère sur des chaînes, c'est-à-dire sur des ensembles d'octets consécutifs, on peut dire aussi sur des zones mémoire.

Il en faut très peu, par exemple, pour que l'instruction LDI qui effectuait déjà un certain nombre d'opérations, se transforme en une instruction de chaîne. Ce pas a été franchi !

Nous serons amenés à distinguer trois groupes d'instructions de chaîne :

- les instructions de transfert,
- les instructions de comparaison(recherche),
- les instructions d'entrées/sorties.

Les mnémoniques de ces instructions se terminent par le suffixe "**R**", indiquant par là une **R**épétition du processus.

Transfert

adressage	instructions		indicateurs
indirect sur chaine	LDIR	LDDR	N,P,H = 0 les autres inchangés

Ces instructions permettent de transférer une zone mémoire d'un endroit à un autre, en partant du début (LDIR) ou de la fin (LDDR). Pour ce faire, les registres suivants sont utilisés :

```
HL ---> adresse zone source
DE ---> adresse zone destination
BC ---> longueur de la zone en octets.
```

A la fin de l'opération, BC égal zéro et HL/DE ont été incrémentés ou décrémentés (LDIR/LDDR) de la valeur contenue initialement dans BC plus 1.
Et c'est un jeu d'enfant d'écrire maintenant :

```
LD      HL,ZONE1
LD      DE,ZONE2
LD      BC,LONG
LDIR
```

pour transférer la zone 1 dans la zone 2, sur une longueur LONG.

Recherche

Ces deux instructions permettent de rechercher une coïncidence entre l'octet contenu dans l'accumulateur et les octets d'une chaîne pointée par HL. Le processus prend fin lorsque la coïncidence a été établie, ou lorsque la fin de la chaîne est atteinte. Comme pour les instructions précédentes, la chaîne peut être scrutée de haut en bas (adresses en progression croissante avec CPIR) ou de bas en haut (CPDR).

adressage	instructions		indicateurs
indirect sur chaine	CPIR	CPDR	C inchangé N = 1 S et H modifiés V=0 si BC=0 Z=1 si egal

Dans le cas de l'instruction CPIR, le registre A est comparé avec l'octet mémoire pointé par HL. BC est alors décrémenté et HL incrémenté. Si les deux valeurs sont égales, l'instruction se termine et Z est mis à 1, sinon le processus continue jusqu'à trouver l'égalité ou la fin de la chaîne. Dans ce cas, BC a atteint zéro et V est mis à zéro.

Exemple : Soit la chaîne hexa 010203040506H implantée à l'adresse mémoire 4000H.

```
LD      HL,4000H
LD      BC,6
LD      A,3
CPIR
```

Il s'agit donc de rechercher la valeur 3 dans la chaîne implantée à l'adresse 4000H. Lorsque le programme est terminé, HL contient 4003H, BC contient 3 et Z est à 1, indiquant que l'octet a été trouvé. P/V n'est pas passé à zéro, puisque la fin de chaîne n'a pas été atteinte.

Entrées/sorties

Ces instructions permettent de dialoguer avec un organe périphérique, non plus avec un octet, mais avec une zone mémoire de 1 à 256 octets.

adressage	instructions		indicateurs
indirect sur chaine	OTIR INIR	OTDR INDR	C inchangé N et Z = 1 les autres quelconques

L'adresse de l'organe périphérique est contenue dans le registre C et l'adresse de la chaîne à émettre ou à recevoir, dans le registre HL. Comme pour les autres instructions de chaîne, l'avant-dernière lettre de la mnémonique indiquera si l'adresse de la chaîne doit s'Incrémenter ou se Décrémenter.

Exemple : Soit à émettre la chaîne ASCII "BONJOUR!" implantée à l'adresse mémoire 4000H, au périphérique d'adresse 7.

```
LD      HL,4000H
LD      C,7
LD      B,8
OTIR
```

FAMILLE 9 : INSTRUCTIONS DE PILE

Nous voici arrivés à la dernière famille, qui n'est pas inconnue de vous, puisque nous avons déjà eu l'occasion d'utiliser très tôt les instructions PUSH et POP dans les exemples des premiers chapitres.

adressage	instructions		indicateurs
indirect 16 bits	PUSH AF	POP AF	non-affectés sauf pour le POP AF
	PUSH BC	POP BC	
	PUSH DE	POP DE	
	PUSH HL	POP HL	
	PUSH IX	POP IX	
	PUSH IY	POP IY	

On peut dire que l'adressage est indirect, puisque l'accès mémoire à la pile s'effectue par l'intermédiaire du registre 16 bits SP. Nous ne nous étendrons pas sur ces instructions, sauf pour signaler qu'elles permettent aussi l'échange des doubles registres :

```
PUSH    IX
PUSH    BC
POP     IX
POP     BC
```

Après empilage de IX et BC, un dépilage est effectué dans le même ordre, ce qui produit une permutation des deux registres IX et BC.

Après avoir empilé toutes ces instructions, tout au long de ce chapitre, il faut maintenant passer au dépilage...
Les exercices suivants sont là pour ça, ne les négligez pas !

EXERCICES PRATIQUES DU CHAPITRE III

Vous êtes maintenant en possession du jeu complet d'instructions vous permettant d'écrire n'importe quel programme en assembleur.

Cette nouvelle série d'exercices n'est là que pour vous aider à faire le point sur ces connaissances nouvelles.

Comme vous pourrez le constater, il n'y a pas de solution unique pour résoudre un problème, et peut-être trouverez-vous des solutions plus astucieuses que celles qui sont proposées.

A vous de trouver des variantes intéressantes ! Et si vous ne trouvez aucune solution, fermez ce livre, et reprenez-le d'ici un jour ou deux, cela devrait aller mieux. Mais surtout, ne vous paniquez pas : çà se soigne !

EXERCICE 3.1 *: Ecrire un programme sous forme de routine réalisant la conversion d'un symbole hexadécimal sous sa forme ASCII, en son équivalent binaire. Exemple : 41H (lettre "A" en ASCII) doit donner OAH.*
Le registre A sera utilisé en entrée et en sortie de la routine pour contenir les valeurs.

EXERCICE 3.2 *: En utilisant la routine ci-dessus, convertir cette fois un nombre hexadécimal de 00 à OFFH contenu dans le registre BC et exprimé en ASCII, en son équivalent binaire. Résultat dans le registre A. Exemple :*

BC = 3941H ---> A = 9AH

EXERCICE 3.3 *: L'instruction CP ne permet de comparer que des nombres de longueur un octet. Ecrire une routine qui compare deux nombres stockés dans les registres HL et DE (comparaison sur 16 bits).*

EXERCICE 3.4 *: Ecrire une routine qui compare maintenant deux zones mémoire pointées par HL et DE, la longueur étant dans C.*

EXERCICE 3.5 *: Ecrire un programme permettant de multiplier le contenu du registre HL par 10. Résultat dans HL.*

CHAPITRE 4

PSEUDO-INSTRUCTIONS ET MACRO-INSTRUCTIONS

Ces nouvelles commandes obéissent aux mêmes règles syntaxiques que les instructions : leur format source possède, lui aussi, les champs étiquette, opération, opérande et commentaire.

Le terme "pseudo" indique que nous aurons affaire à des "espèces" d'instructions (mais pas tout-à-fait) bien qu'étant parfois très proche de... et le terme "macro", que se sont tout bonnement des sortes d'instructions "géantes"...

Les ***pseudo-instructions***, appelées aussi ***directives***, sont des commandes s'adressant, non au microprocesseur, comme le faisaient les instructions, mais au programme assembleur lui-même.
Ces commandes permettent d'imposer certaines règles concernant l'édition du listing, l'implantation du programme, la définition de constantes ou de données mémoire, tout en offrant certaines facilités d'écriture du code source.

Les ***macro-instructions***, quant à elles, élèvent le degré d'évolution du langage d'assemblage, en abrégeant l'écriture du code source par l'emploi de mots nouveaux, réalisant des fonctions pré-définies par le programmeur.
Lorsqu'une séquence d'instructions devient répétitive dans un programme (oh que c'est lassant...!), il suffit de la déclarer comme macro-instruction en lui affectant un nom.
A chaque fois que ce nom sera invoqué par la suite, la séquence d'instructions correspondante sera produite en place de ce nom. L'assembleur, dans ce cas, porte le nom de ***macro-assembleur***.

Mais reprenons du début.

LES DIRECTIVES (OU PSEUDO-INSTRUCTIONS)

Nous n'allons pas les décrire toutes (il y en a tous les jours de nouvelles, c'est à croire qu'elles font des petits...), mais nous nous bornerons seulement à celles qui sont les plus courantes.

Tout d'abord, celles qui sont impératives dans tout assembleur digne de ce nom :

FIXATION DE L'ORIGINE : ORG

Cette directive permet de fixer l'adresse d'origine du programme - ou d'une portion de programme - en mémoire. Son opérande indique l'adresse mémoire à laquelle doit être généré le code objet qui suit.

Exemple :

```
        ORG     1000H
        LD      A,2
```

l'instruction LD A,2 sera implantée à partir de l'adresse 1000H de la mémoire.

Si cela est nécessaire, plusieurs commandes ORG peuvent être utilisées :

```
        ORG     0
        JP      PROG1
        ORG     8
        JP      PROG2
          .....
```

En l'absence de directive ORG, le code objet est généré à partir de l'adresse 0. La zone argument peut être une expression, à condition qu'elle soit entièrement définie lors de la passe 1 de l'assembleur.

FIN DU PROGRAMME SOURCE : END

C'est la dernière ligne source du programme. Elle a pour fonction d'indiquer à l'assembleur que la fin du programme est atteinte.

Son opérande précise l'adresse à laquelle le programme devra être lancé lors de son chargement, c'est-à-dire l'adresse de la première instruction à exécuter (et qui n'est pas forcément la première).

Exemple :

```
ROUT    LD      A,4
        RLCA
        AND     B
        RET
DEB     LD      SP,4000H
          ......
        END     DEB
```

Après chargement de ce programme en mémoire, l'instruction située à l'adresse DEB sera exécutée.

Mais l'opérande n'est pas forcément une étiquette, et peut prendre la forme d'une valeur d'adresse hexadécimale ou décimale, même si elle se trouve en dehors du programme :

```
        END     100H
```

DEFINITION D'EQUIVALENCE : EQU et DEFL

Lorsque l'on préfère manipuler des symboles - toujours plus explicites - que des valeurs, dans un programme, on emploie la directive EQU qui ASSIGNE une valeur déterminée et immuable à un symbole.

Exemple :

```
ABUF      EQU     4000H
FINBUF    EQU     4FFFH
LONG      EQU     FINBUF-ABUF+1
N         EQU     4
          SET     N,A
            ......
```

Mais lorsqu'une équivalence est définie, elle ne peut plus être modifiée durant l'exécution du programme. Il y a un moyen pourtant : en employant la directive DEFL à la place de l'EQU, l'assembleur ne produit pas d'erreur de double assignation :

```
N         DEFL     4
          SET      N,A
N         DEFL     7
          BIT      N,A
            ......
```

DEFINITION DE DONNEES : DEFB, DEFW et DEFM

On les appelle aussi : BYTE, WORD et ASCII dans certains assembleurs. Contrairement aux précédentes, ces directives produisent du code objet.

Dans un programme, il n'y a pas forcément que des instructions : il peut aussi y avoir des données. Souvenez-vous du texte "BONJOUR" des exemples du chapitre I (Oh ! que c'est loin !).

Nous pourrons maintenant le définir ainsi :

```
DEFM     'BONJOUR'
```

et l'assembleur produira la chaîne ASCII correspondant à ce texte (DEFM : **DEF**ine **M**essage).

De la même façon, nous pouvons définir un octet (byte en Anglais) ou un mot (word) de deux octets :

```
          ORG      4000H
OCT       DEFB     41H              ; DEFinition Byte
MOT       DEFW     0AFFH            ; DEFinition Word
DEB       LD       A,(OCT)
          LD       BC,(MOT)
```

Le registre A contiendra la valeur 41H et le registre BC contiendra la valeur 0AFFH.

Attention : pas de confusion avec :

```
LD       BC,MOT
```

qui signifie que BC contiendra l'adresse mémoire correspondant au label mot, c'est-à-dire 4001H.

Lorsque la valeur à définir est en ASCII et est imprimable, nous pouvons écrire :

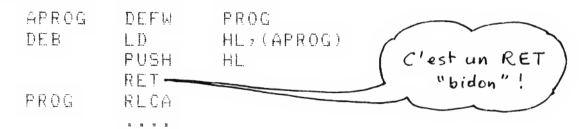
```
OCT      DEFB     'A'
```

Bien entendu, les labels peuvent aussi participer à la danse !

```
LG       DEFB     FBUF-BUF
```

Attention : La valeur résultante ne doit pas dépasser OFFH dans ce cas.

```
APROG    DEFW     PROG
DEB      LD       HL,(APROG)
         PUSH     HL
         RET
PROG     RLCA
         ....
```



Mais prenons un exemple concret: soit à émettre, vers un organe périphérique, un texte en ASCII. L'appel de la routine VISU provoquera l'envoi d'un caractère du message vers l'écran vidéo, par exemple. Nous écrirons :

```
DEB      LD       HL,ATEXT           ; pointe debut texte
SUIT     LD       A,(HL)
         OR       A                  ; pour tester Z
         JR       Z,FIN              ; si Z --> arrêt
         CALL     VISU               ; emission 1 caract
         INC      HL                 ; +1 sur adr. texte
         JR       SUIT               ; suite
ATEXT    DEFM     'OH! COMBIEN DE MARINS'
         DEFB     ODH
         DEFM     'COMBIEN DE CAPITAINES...'
         DEFB     0
```

Le texte ci-dessus défini est composé de deux lignes de caractères ASCII, séparées par le code hexa ODH qui correspond au retour ligne (voir code ASCII en annexe). Comme ce code n'est pas imprimable, il faut le définir sous sa forme hexadécimale (ou décimale). Le texte est terminé, dans notre exemple, par le caractère zéro qui en indique la fin (cela aurait pu être un tout autre code).

Ce programme émet donc le message, caractère par caractère, par l'intermédiaire de la routine VISU qui est spécifique au périphérique (on appelle cela un DRIVER), jusqu'à détection du zéro de fin de texte. Un branchement est effectué à l'adresse FIN, qui permet de sortir de la boucle (non représentée dans l'exemple).

Vous remarquerez que le texte doit être délimité par deux caractères d'apostrophe. Lorsque ce caractère doit figurer à l'intérieur du texte, il est nécessaire de le répéter dans le code source.

Exemple :

```
DEFM     'L'ASSEMBLEUR, C'EST FACILE'
```

provoquera une erreur à l'assemblage et devra être écrit :

```
DEFM     'L''ASSEMBLEUR, C''EST FACILE'
```

Notons enfin que certains assembleurs admettent la forme multi-arguments dans le DEFB et le DEFW :

```
DEFB    0,1,0ACH,7,ADR,'*'
```

RESERVATION D'ESPACE : DEFS

Cette directive permet de réserver la place pour un certain nombre d'octets en mémoire :

```
DEFS    257
```

réservera un emplacement de 257 octets dans le programme. En pratique, le pointeur d'adresse est incrémenté de cette valeur par l'assembleur.

Attention aux surprises : l'espace réservé n'est pas forcément mis à zéro...

```
OCT1    DEFB    4
        DEFS    10
OCT2    DEFB    2
```

Si OCT1 est à l'adresse mémoire 1F4H par exemple, OCT2 sera à l'adresse 1FFH.

Les directives que nous allons étudier maintenant ne sont présentes que dans les assembleurs "musclés" et les macro-assembleurs. Leurs noms varient d'un assembleur à l'autre, mais le principal est de savoir qu'elles existent et ce qu'elles font.

DIRECTIVES AGISSANT SUR L'EDITION DU LISTING

PAGE (ou $EJECT) forcera le listing d'assemblage à se poursuivre sur la page suivante (saut de page). Il est parfois possible de définir, du même coup, le nombre de lignes par page :

```
        PAGE    66
ADR     LD      SP,40FFH
          ......
```

TITLE (ou $TITLE), permet d'imprimer systématiquement en haut de chaque page du listing, un texte donné :

```
TITLE   "PROGRAMME DE GESTION. JUIN 1980"
```

SUBTTL (ou $STITLE) est identique au précédent, mais édite un sous-titre après le texte du titre.

NOLIST et ***LIST*** permettent d'interrompre ou de reprendre l'édition du listing :

```
        LD      SP,4FFFH
        NOLIST
; Cette portion de programme
; ne sera pas editée
```

```
LIST
PUSH    AF
   ......
```

DIRECTIVES D'ASSEMBLAGE CONDITIONNEL

Il arrive fréquemment qu'un programme possède des séquences optionnelles qui doivent ou non être assemblées selon la configuration désirée.

On peut, par exemple, envisager un programme dont la configuration "mini" permet d'éditer des résultats sur un écran vidéo, alors que la configuration "maxi" les édite sur l'écran vidéo et sur une imprimante.

Afin d'éviter d'avoir à écrire deux programmes presque identiques, il suffira de placer, en tête du code source :

```
MINI    EQU     0
MAXI    EQU     1
```

précédé du choix de la configuration :

```
CONFIG  EQU     MAXI
```

ou :

```
CONFIG  EQU     MINI
```

pour pouvoir exploiter ensuite les directives permettant d'assembler ou non certaines séquences du programme. Ces directives vous rappelleront peut-être quelques souvenirs : IF et ELSE ?

Le test de la valeur d'un symbole indiquera si les lignes qui suivent doivent être assemblées ou non :

```
IF      symbole est vrai...
on assemble ceci
ELSE    (sinon)
on assemble cela
ENDIF
```

(le ENDIF indique la fin de la séquence source faisant l'objet d'un assemblage conditionnel).

Dans l'exemple précédent, nous écrirons :

```
IF      CONFIG = MAXI
......                        ; edition imprimante
......
ENDIF
.......                       ; edition video
```

le ELSE n'est, dans ce cas, d'aucune utilité.

Autre exemple :

```
INCDEC  EQU     1
         ......
         ......
        IF      INCDEC = 1
        INC     A
        ELSE
        DEC     A
        ENDIF
```

En modifiant la première ligne du programme, l'assembleur produira l'instruction INC A ou DEC A selon le résultat du test.

DIRECTIVE DE REPETITION

Les lignes sources comprises entre les directives ***REPEAT*** (ou RPT) et ***ENDR*** sont répétées un nombre de fois précisé dans l'opérande :

```
        REPEAT  3
        DEFB    2
        ENDR
```

produira :

```
        DEFB    2
        DEFB    2
        DEFB    2
```

De même, en écrivant :

```
N       DEFL    0
        REPEAT  3
        DEFB    N
N       DEFL    N+1
        ENDR
```

l'assembleur produira :

```
        DEFB    0
        DEFB    1
        DEFB    2
```

DIRECTIVE DE CHARGEMENT D'UN MODULE SOURCE

La directive ***INCLUDE*** (ou LOAD), lorsqu'elle est rencontrée, provoque l'appel et l'inclusion dans le code source d'un autre programme source issu d'un périphérique (disque en particulier) et dont le nom figure en opérande de la directive :

```
        LD      SP,0FFFFH
        INCLUDE PROG1
            ......
```

Le module source PROG1 viendra s'insérer dans le code source, à l'endroit de l'appel.

DECLARATION ET DEFINITION DE SYMBOLES EXTERNES

Ce sont les directives :

EXTERNAL (ou EXT ou EXTRN)
GLOBAL (ou ENTRY ou PUBLIC)

Vous devez maintenant savoir que si l'on fait référence, dans un programme, à un symbole absent de ce même programme, l'assembleur signale systématiquement une erreur.

D'un autre côté, il peut être intéressant d'assembler séparément plusieurs morceaux d'un programme, pour les réunir par la suite en un seul ensemble.

Nous verrons que cela peut être rendu possible grâce à un utilitaire nommé l'***éditeur de liens***.

Mais, si tel est le cas, comment écrire par exemple :

```
        CALL            ROUT1
```

alors que ROUT1 se trouve dans un autre "morceau" du programme source, et cela, sans provoquer d'erreur d'assemblage ?

Il suffira de déclarer, dans notre cas, que ROUT1 est externe au programme actuel :

```
        EXTERNAL        ROUT1
                ........
        CALL            ROUT1
```

Dans le module où se trouve cette routine, il sera aussi nécessaire d'indiquer que le symbole ROUT1 peut être appelé par un module externe :

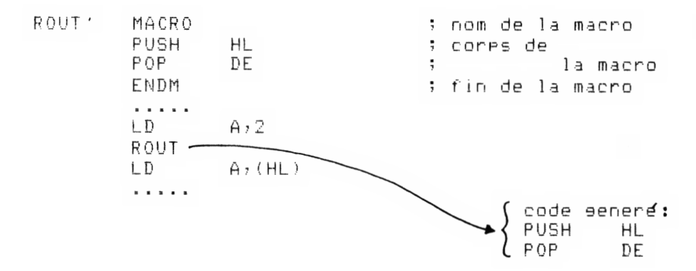
```
        GLOBAL          ROUT1
ROUT1   LD              A,1
                ......
```

Lorsque ces deux conditions sont réunies, l'assembleur n'y voit que du feu... et ne signale pas d'erreur.

LES MACRO-INSTRUCTIONS

Pour définir une macro, on utilise deux directives : ***MACRO*** et ***ENDM***, qui délimitent la liste des lignes de code source formant le corps de la macro.

Exemple :



Dans cet exemple, la macro portant le nom ROUT est formée des deux instructions PUSH HL et POP DE. La simple invocation de ce nom en zone opération du code source provoque la génération, par l'assembleur, de la liste des instructions définies précédemment, qui viennent s'inclure dans le programme.

Attention : Il ne faut pas faire de confusion entre les macro-instructions et les sous-programmes. Les premières ont pour seul but de simplifier et d'abréger l'écriture du code source, mais elles n'ont aucune incidence sur la longueur du code objet comme le font les sous-programmes.

L'utilisation des macros serait relativement réduite s'il n'était pas possible de les paramétrer...

Un exemple nous éclairera, à ce sujet :

```
ROUT    MACRO   P
        LD      &P,2
        AND     (HL)
        ENDM
```

Dans la macro définie ci-dessus, P représente le nom (quelconque) d'un paramètre qu'il faudra substituer, à l'endroit du signe "&" au symbole fourni à l'appel de la macro.

Si, par exemple, nous écrivons : `ROUT A`

l'assembleur produira :

```
LD      A,2
AND     (HL)
```

alors qu'en écrivant : `ROUT B`

il aurait produit :

```
LD      B,2
AND     (HL)
```

On peut évidemment "passer" plusieurs paramètres :

```
ROUT     MACRO   P,X,Y
LABEL&X  LD      &P,&Y
         AND     (HL)
         INC     HL
         JR      NZ,LABEL&X
         ENDM
```

et l'appel :

```
ROUT     A,1,0AH
```

produira la séquence :

```
LABEL1   LD      A,0AH
         AND     (HL)
         INC     HL
         JR      NZ,LABEL1
```

Super-macro-génial !

Mais imaginez un peu la définition suivante :

```
MPROG   MACRO
LAB1    LD      A,(HL)
        AND     A
        JR      NZ,LAB1
        ENDM
```

Que se passera t-il si cette macro est appelée plus d'une fois dans le programme - ce qui ne manquera pas de se produire - ? Le label LAB1 sera généré plus d'une fois également, et l'assembleur, intransigeant devant une telle ignominie, signalera une erreur de double définition d'étiquette !

Il faudra alors faire usage d'une nouvelle directive indiquant que l'utilisation de ce label est interne ou local à la macro, et nous écrirons :

```
MPROG   MACRO
        LOCAL   LAB1
LAB1    LD      A,(HL)
        ......
```

et le problème sera résolu !

Dernier point au sujet des macros :

Il peut être souhaitable d'en "sortir" prématurément à la suite, par exemple, d'un test d'une condition IF.

La directive ***EXITM*** sera utilisée dans ce but :

```
PROG    MACRO   N1
        LD      A,1
        IF      &N1 = 0
        EXITM
        AND     (HL)
        ENDIF
         .....
```

N'est-ce pas là une bonne occasion pour faire un EXIT du chapitre 4 ?

CHAPITRE 5

TECHNIQUES ET PRATIQUE DE L'ASSEMBLEUR

Dans ce chapitre essentiellement pratique, nous allons décrire, au travers d'exemples, quelques-unes des principales techniques de programmation mises en oeuvre implicitement dans tout programme écrit en assembleur.

Ainsi, nous serons amenés à parler de boucles, de tables, de recherche, de sous-programmes, de calculs, de conversions, de gestion des entrées/sorties... et nous nous apercevrons que nous utilisons parfois ces techniques aussi facilement que Monsieur Jourdain parlait en prose.

La programmation - à l'inverse des mathématiques - n'est pas encombrée de noms pompeux, signifiant que pour résoudre tel ou tel problème, vous devez vous appuyer sur Thalès, en passant par Kuriosevic, et que grâce au théorème de Stroupf et Williams, vous êtes enfin parvenu à démontrer que le lemme de Plucker modifié par Schwartz pouvait dans certains cas...

Non ! La programmation est une chose personnelle. Lorsque vous avez écrit un programme, vous avez le sentiment que c'est bien vous qui l'avez écrit.

En fait, il n'y a pas de méthode rigide de programmation, mais un ensemble de moyens "ouverts", mis à la disposition du programmeur qui les adaptera plus ou moins au type de problème qu'il doit résoudre.

Ainsi, deux programmes écrits séparément par deux personnes peuvent faire appel à des techniques différentes, et pourtant résoudre les mêmes problèmes. L'un s'exécutera peut-être plus rapidement que l'autre, ou sera plus encombrant en mémoire, c'est là qu'intervient l'art de la programmation, que l'on ne peut apprendre que par l'expérience.

Ces différentes techniques étant plus ou moins imbriquées et dépendantes les unes des autres, nous nous bornerons dans ce chapitre, à décrire et à commenter de petits programmes d'usage général. Lorsque vous aurez compris le mécanisme de chacun d'eux, vous serez sur la bonne voie !

```
                 00100 ; ROUTINE DE CONVERSION
                 00110 ; HEXA ---> 1 OCTET ASCII
                 00120 ; ENTREE: (A)= DIGIT HEXA
                 00130 ; SORTIE: (A)= OCTET ASCII
                 00140 ;
8000             00145        ORG     8000H
8000 C690        00150 HEXASC ADD     A,90H
8002 27          00160        DAA
8003 CE40        00170        ADC     A,40H
8005 27          00180        DAA
8006 C9          00190        RET
8000             00200        END     HEXASC
00000 TOTAL ERRORS
```

Le but de ce programme est de convertir les valeurs hexadécimales 0 à 9 en codes ASCII 30 à 39, et les valeurs hexadécimales 0A à 0F en codes ASCII 41 à 46 (voir tableau ASCII en annexe).

Deux exemples nous aideront à mieux comprendre :

```
valeurs hexa:                   0A   !       09
--------------------------------------+----------
                                90   !       90
addition:                     + 0A   !     + 09
--------------------------------------+----------
                                9A   !       99
action DAA:                   + 66   !     + 00
--------------------------------------+----------
                             (1) 00   !       99
addition avec carry:          + 40   !     + 40
                              +  1   !     +  0
--------------------------------------+----------
                                41   !     0 D9
action DAA:                   + 00   !     + 60
--------------------------------------+----------
                                41   !     1 39
```

Et dîtes-vous que si cela marche pour ces deux cas, cela marchera pour les autres !

```
                  010 ; CONVERSION BINAIRE-DECIMALE
                  020 ; EN ASCII SUR 1 A 5 CHIFFRES
                  030 ; ENTREE: (HL): ADR RESULTAT
                  040 ;          (DE): VALEUR BINAIRE
                  050 ;
8000              060           ORG     8000H
8000 01F0D8       070 BINDC5    LD      BC,-10000      ; 5 CHIFFRES
8003 CD1B80       080           CALL    CBD            ;
8006 0118FC       090 BINDC4    LD      BC,-1000       ; 4 CHIFFRES
8009 CD1B80       100           CALL    CBD            ;
800C 019CFF       110 BINDC3    LD      BC,-100        ; 3 CHIFFRES
800F CD1B80       120           CALL    CBD            ;
8012 01F6FF       130 BINDC2    LD      BC,-10         ; 2 CHIFFRES
8015 CD1B80       140           CALL    CBD            ;
8018 01FFFF       150 BINDC1    LD      BC,-1          ; 1 CHIFFRE
                  160 ;
                  170 ; ROUTINE DE CONVERSION
                  180 ;
801B 3E2F         190 CBD       LD      A,'0'-1        ; 0 ASCII-1
801D E5           200           PUSH    HL             ; ADR RANGT
801E D5           210           PUSH    DE             ; VALEUR BIN
801F E1           220           POP     HL             ; DE--->HL
8020 3C           230 CBD1      INC     A              ; UNITE+1
8021 09           240           ADD     HL,BC          ;
8022 38FC         250           JR      C,CBD1         ;
8024 ED42         260           SBC     HL,BC          ; EQUILIB HL
8026 EB           270           EX      DE,HL          ;
8027 E1           280           POP     HL             ; ADR RANGT
8028 77           290           LD      (HL),A         ; RESULTAT
8029 23           300           INC     HL             ; SUITE
802A C9           310           RET                    ; ET RETOUR
8000              320           END     BINDC5
00000 TOTAL ERRORS
```

Cette routine assure la conversion d'un nombre de 16 bits contenu dans le registre DE, en une chaîne de caractères ASCII (s'il vous plaît !) exprimée sous forme décimale, et dont l'adresse est passée par l'intermédiaire du registre HL.

Pour utiliser la routine, il suffit de faire :

```
LD      HL,4000H        ; adresse chaine
LD      DE,0FB3AH       ; nombre à convertir
CALL    BINDC5          ; appel conversion
```

Après exécution, la chaîne rangée en 4000H contiendra :

```
4000H:  36      (6 ASCII)
4001H:  34      (4)
4002H:  33      (3)
4003H:  31      (1)
4004H:  34      (4)
```

c'est-à-dire le nombre 64314.

Autre exemple :

```
LD      DE,00ABH        ; nombre à convertir
CALL    BINDC5          ; appel conversion
```

La chaîne résultante sera : 30 30 31 37 31, c'est-à-dire 00171. Dans le cas présent, il était possible de faire :

```
CALL    BINDC3
```

et la chaîne aurait été : 31 37 31, soit le nombre 171.

En examinant ce programme, nous voyons que celui-ci utilise une sous-routine CBD qui est appelée autant de fois que le nombre de chiffres désirés.

Remarquez que le dernier appel n'est pas fait par un CALL.

Il aurait été plus coûteux de faire :

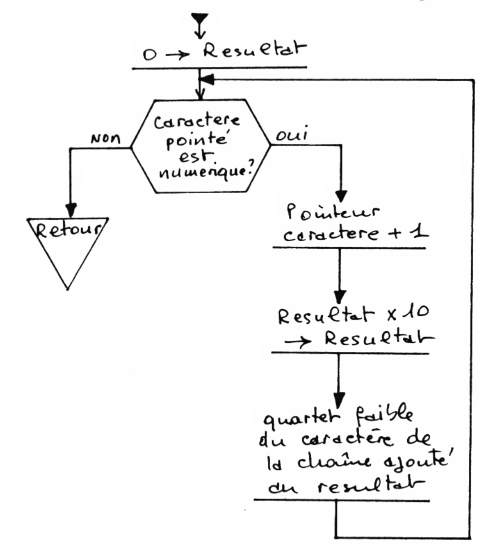
```
BINDC1  LD      BC,-1
        CALL    CBD
        RET
```

De plus, cela ne présentait aucun intérêt.

Le principe de calcul est simple : on compte par soustraction, le nombre de fois qu'il y a 10000 dans le nombre, puis successivement 1000, 100, 10 et 1 et pour chaque tranche ce nombre de fois est ajouté à la valeur 2FH pour donner le chiffre des dix mille, mille... de valeur comprise entre 30 et 39 (0 à 9 en ASCII).

```
00130 ; CONVERSION DECIMALE - BINAIRE
00140 ; ENTREE (HL): ADR CHAINE ASCII
00150 ;        (DE): RESULTAT BINAIRE
00160 ; FIN DE CONVERSION SUR CARACTERE
                   00170 ; ASCII NON NUMERIQUE
                   00180 ;
8000               00190          ORG    8000H
8000 110000        00200 DECBIN   LD     DE,0      ; 0--> RESULTAT
8003 7E            00210 DCB1     LD     A,(HL)    ; CARAC CHAINE
8004 FE30          00220          CP     '0'       ; NUMERIQUE?
8006 D8            00230          RET    C         ; SINON: FIN
8007 FE3A          00240          CP     '9'+1     ; TEST SI > 9
8009 D0            00250          RET    NC        ; SI OUI: FIN
800A 23            00260          INC    HL        ; ADR CAR SUIVANT
800B E5            00270          PUSH   HL        ; PROTEGEE
800C 62            00280          LD     H,D       ;
800D 6B            00290          LD     L,E       ; DE--> HL
800E 19            00300          ADD    HL,DE     ;
800F 29            00310          ADD    HL,HL     ;
8010 19            00320          ADD    HL,DE     ;
8011 29            00330          ADD    HL,HL     ; 10 * DE --> HL
8012 D630          00340          SUB    '0'       ; QUARTET FAIBLE
8014 5F            00350          LD     E,A       ;
8015 1600          00360          LD     D,0       ;
8017 19            00370          ADD    HL,DE     ; CUMUL RESULTAT
8018 EB            00380          EX     DE,HL     ; --> DE
8019 E1            00390          POP    HL        ; POINTEUR CAR
801A 18E7          00400          JR     DCB1      ; ET SUITE
8000               00410          END    DECBIN
00000 TOTAL ERRORS
```

Ce programme exécute la conversion inverse du précédent. La chaîne décimale ASCII rangée à l'adresse pointée par HL et terminée par un caractère non numérique, est convertie en une valeur binaire fournie en sortie, dans le registre DE.



Si la zone mémoire 4000H contient : 36 34 33 31 34 00 (nombre 64314), l'appel :

```
LD      HL,4000H
CALL    DECBIN
```

rendra la valeur OFB3AH dans le registre DE.

Le programme teste tout d'abord si chaque chiffre de la chaîne est compris entre 30 et 39, c'est-à-dire 0 et 0 ASCII, auquel cas c'est un caractère d'arrêt (fin de chaîne).

A chaque tranche (puissance de dix), le résultat courant est multiplié par 10 et la valeur binaire (poids faibles) de chaque chiffre de la chaîne lui est ajoutée.

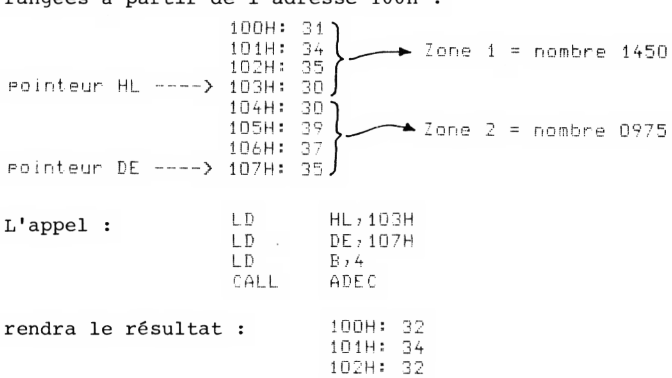
Dans notre exemple, la décomposition sera :

```
DE = 0
DE * 10 = 0 ---------> + 6 ----> DE -----> = 6
DE * 10 = 60 --------> + 4 ----> DE -----> = 64
DE * 10 = 640 -------> + 3 ----> DE -----> = 643
DE * 10 = 6430 ------> + 1 ----> DE -----> = 6431
DE * 10 = 64310 -----> + 4 ----> DE -----> = 64314
```

Bien entendu, comme toutes les opérations se font en binaire, le résultat 64314 est exprimé en binaire lui aussi.

```
                    00100 ; ADDITION DECIMALE ENTIERE
                    00110 ; DE 2 ZONES ASCII
                    00120 ; ENTREE (HL): POINTE FIN ZONE 1
                    00130 ;        (DE): POINTE FIN ZONE 2
                    00140 ;         (B): LONGUEUR
                    00150 ; RESULTAT DANS ZONE 1
                    00160 ;
0025                00170         ORG     25H
0025 4E             00180 ADEC    LD      C,(HL)   ; CARACT ZONE 1
0026 1A             00190         LD      A,(DE)   ; CARACT ZONE 2
0027 81             00200         ADD     A,C      ; CAR1 + CAR2
0028 D630           00210         SUB     '0'      ;
002A FE3A           00220         CP      3AH      ; RETENUE?
002C 3805           00230         JR      C,NRET   ; NON
002E D60A           00240         SUB     0AH      ;
0030 2B             00250         DEC     HL       ; REPORT
0031 34             00260         INC     (HL)     ; RETENUE SUR
0032 23             00270         INC     HL       ; CAR SUIVANT
0033 77             00280 NRET    LD      (HL),A   ; RANGEMENT
0034 2B             00290         DEC     HL       ; ON PASSE AUX
0035 1B             00300         DEC     DE       ; CARACT SUIVANTS
0036 10ED           00310         DJNZ    ADEC     ; SUITE
0038 C9             00320         RET              ; SINON RETOUR
0025                00330         END     ADEC
00000 TOTAL ERRORS
```

Ce programme exécute l'addition décimale de deux chaînes ASCII de même longueur. Soit par exemple ces deux chaînes rangées à partir de l'adresse 100H :



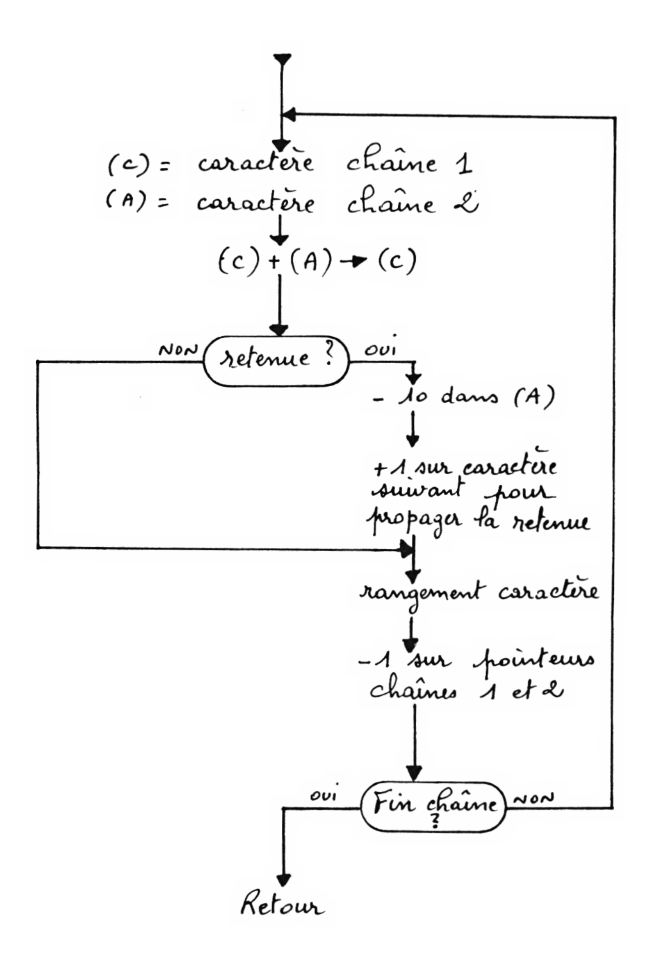
L'appel :

```
LD      HL,103H
LD      DE,107H
LD      B,4
CALL    ADEC
```

rendra le résultat :

```
100H: 32
101H: 34
102H: 32
103H: 35
```

correspondant au nombre 2425 en ASCII.



Pour bien suivre le déroulement de ce programme, nous vous proposons, ci-après, le listing décrivant en pas-à-pas l'exécution de chaque instruction et le contenu dynamique des registres.

LOC	INST	MNEM	OPER	SP	RF	RA	RB	RC	RD	RE	RH	RL
0025	4E	LD	C,(HL)	0204	AB	32	04	30	01	07	01	03
0026	1A	LD	A,(DE)	0204	AB	35	04	30	01	07	01	03
0027	81	ADD	A,C	0204	20	65	04	30	01	07	01	03
0028	D630	SUB	A,30	0204	22	35	04	30	01	07	01	03
002A	FE3A	CP	3A	0204	BB	35	04	30	01	07	01	03
002C	3805	JR	C,07	0204	BB	35	04	30	01	07	01	03
0033	77	LD	(HL),A	0204	BB	35	04	30	01	07	01	03
0034	2B	DEC	HL	0204	BB	35	04	30	01	07	01	02
0035	1B	DEC	DE	0204	BB	35	04	30	01	06	01	02
0036	10ED	DJNZ	EF	0204	BB	35	03	30	01	06	01	02
0025	4E	LD	C,(HL)	0204	BB	35	03	35	01	06	01	02
0026	1A	LD	A,(DE)	0204	BB	37	03	35	01	06	01	02
0027	81	ADD	A,C	0204	28	6C	03	35	01	06	01	02
0028	D630	SUB	A,30	0204	2A	3C	03	35	01	06	01	02
002A	FE3A	CP	3A	0204	2A	3C	03	35	01	06	01	02
002C	3805	JR	C,07	0204	2A	3C	03	35	01	06	01	02
002E	D60A	SUB	A,0A	0204	22	32	03	35	01	06	01	02
0030	2B	DEC	HL	0204	22	32	03	35	01	06	01	01
0031	34	INC	(HL)	0204	20	32	03	35	01	06	01	01
0032	23	INC	HL	0204	20	32	03	35	01	06	01	02
0033	77	LD	(HL),A	0204	20	32	03	35	01	06	01	02
0034	2B	DEC	HL	0204	20	32	03	35	01	06	01	01

LOC	INST	MNEM	OPER	SP	RF	RA	RB	RC	RD	RE	RH	RL
0035	1B	DEC	DE	0204	20	32	03	35	01	05	01	01
0036	10ED	DJNZ	EF	0204	20	32	02	35	01	05	01	01
0025	4E	LD	C,(HL)	0204	20	32	02	35	01	05	01	01
0026	1A	LD	A,(DE)	0204	20	39	02	35	01	05	01	01
0027	81	ADD	A,C	0204	28	6E	02	35	01	05	01	01
0028	D630	SUB	A,30	0204	2A	3E	02	35	01	05	01	01
002A	FE3A	CP	3A	0204	2A	3E	02	35	01	05	01	01
002C	3805	JR	C,07	0204	2A	3E	02	35	01	05	01	01
002E	D60A	SUB	A,0A	0204	22	34	02	35	01	05	01	01
0030	2B	DEC	HL	0204	22	34	02	35	01	05	01	00
0031	34	INC	(HL)	0204	20	34	02	35	01	05	01	00
0032	23	INC	HL	0204	20	34	02	35	01	05	01	01
0033	77	LD	(HL),A	0204	20	34	02	35	01	05	01	01
0034	2B	DEC	HL	0204	20	34	02	35	01	05	01	00
0035	1B	DEC	DE	0204	20	34	02	35	01	04	01	00
0036	10ED	DJNZ	EF	0204	20	34	01	35	01	04	01	00
0025	4E	LD	C,(HL)	0204	20	34	01	32	01	04	01	00
0026	1A	LD	A,(DE)	0204	20	30	01	32	01	04	01	00
0027	81	ADD	A,C	0204	20	62	01	32	01	04	01	00
0028	D630	SUB	A,30	0204	22	32	01	32	01	04	01	00
002A	FE3A	CP	3A	0204	BB	32	01	32	01	04	01	00
002C	3805	JR	C,07	0204	BB	32	01	32	01	04	01	00

LOC	INST	MNEM	OPER	SP	RF	RA	RB	RC	RD	RE	RH	RL
0033	77	LD	(HL),A	0204	BB	32	01	32	01	04	01	00
0034	2B	DEC	HL	0204	BB	32	01	32	01	04	00	FF
0035	1B	DEC	DE	0204	BB	32	01	32	01	03	00	FF
0036	10ED	DJNZ	EF	0204	BB	32	00	32	01	03	00	FF
0038	C9	RET		0206	BB	32	00	32	01	03	00	FF

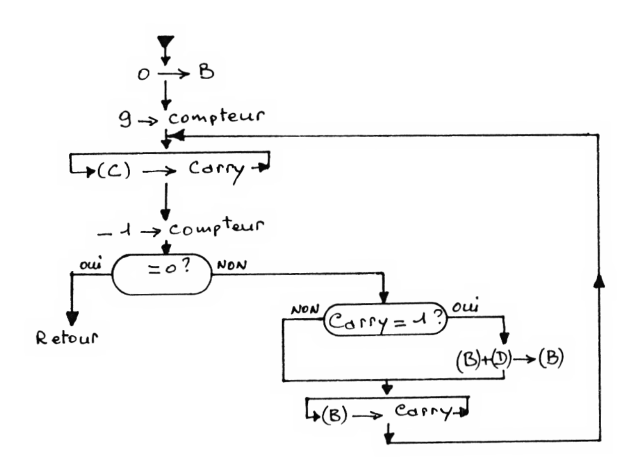
```
                   00100 ; MULTIPLICATION BINAIRE
                   00110 ; (D) * (C) ---> (BC)
                   00120 ;
0000 0600          00130 MULT    LD      B,0      ;
0002 1E09          00140         LD      E,9      ; COMPTEUR BITS
0004 79            00150 MU0     LD      A,C      ;
0005 1F            00160         RRA              ;
0006 4F            00170         LD      C,A      ;
0007 1D            00180         DEC     E        ; NBRE DE BITS -1
0008 C8            00190         RET     Z        ; RETOUR SI = 0
0009 78            00200         LD      A,B      ;
000A 3001          00210         JR      NC,MU1   ;
000C 82            00220         ADD     A,D      ;
000D 1F            00230 MU1     RRA              ; DECALAGE BIT
000E 47            00240         LD      B,A      ;
000F 18F3          00250         JR      MU0      ;
0000               00260         END     MULT
00000 TOTAL ERRORS
```

Comme nous le savons maintenant, le Z80 ne possède pas d'instruction de multiplication. Qu'à cela ne tienne, cette routine permettra de s'en passer !

```
        LD      D,2BH
        LD      C,0A0H
        CALL    MULT
```

Le registre BC contiendra la valeur 1AE0H.
L'intérêt de la méthode employée est qu'elle est pratiquement constante en durée, quelles que soient les valeurs à multiplier.

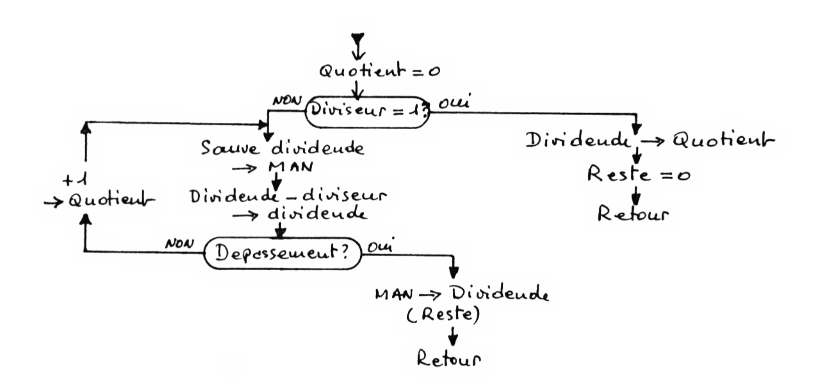
Le processus de calcul est fort simple et est représenté par l'organigramme ci-dessous :



Mais, passons maintenant à la division qui, elle aussi, ne figure pas dans le jeu d'instructions du microprocesseur.

```
                00100 ; DIVISION BINAIRE
                00110 ; (HL) / (DE)
                00120 ; EN SORTIE: (BC) = QUOTIENT
                00130 ;                (HL) = RESTE
                00140 ;
0000 010000     00150 DIV     LD      BC,0      ; INIT RESULTAT
0003 7A         00160         LD      A,D       ; POIDS FORTS
0004 FE00       00170         CP      0         ; QUOTIENT NUL?
0006 200B       00180         JR      NZ,DIV1   ; NON.
0008 7B         00190         LD      A,E       ; OUI. POIDS
0009 FE01       00200         CP      1         ; FAIBLES = 1?
000B 2006       00210         JR      NZ,DIV1   ; NON.
000D 44         00220         LD      B,H       ; OUI. HL-->BC
000E 4D         00230         LD      C,L       ;
000F 210000     00240         LD      HL,0      ; RESTE = 0
0012 C9         00250         RET               ; SORTIE
0013 222500     00260 DIV1    LD      (MAN),HL  ; SAUVE RESTE
0016 7D         00270         LD      A,L       ;
0017 93         00280         SUB     E         ;
0018 6F         00290         LD      L,A       ;
0019 7C         00300         LD      A,H       ;
001A 9A         00310         SBC     A,D       ;
001B 67         00320         LD      H,A       ;
001C 3803       00330         JR      C,DIV2    ;
001E 03         00340         INC     BC        ; QUOTIENT + 1
001F 18F2       00350         JR      DIV1      ;
0021 2A2500     00360 DIV2    LD      HL,(MAN)  ; RESTE --> HL
0024 C9         00370         RET               ; ET RETOUR
0025            00380 MAN     DEFS    2         ; MANOEUVRE
0000            00390         END     DIV
00000 TOTAL ERRORS
```

La méthode employée ici est la plus simple, et procède par soustractions successives, comme il est facile de le voir sur l'organigramme ci-dessous. Le programme pourrait être simplifié en employant l'instruction SBC HL,DE en place des lignes 270 à 320, mais il faudrait la faire précéder d'un OR A afin de ne pas soustraire Carry...



Le programme qui suit met en oeuvre les techniques de recherche en table, de boucle et d'appel d'entrée/sortie. Le principe est le suivant : on entre un caractère numérique au clavier, et le programme fournit dans le registre A la codification en code Morse de ce caractère. L'entrée d'un caractère au clavier sera effectuée par l'appel de la routine CLAV.

Dans le cas où un caractère n'est pas dans la table, un message d'erreur est envoyé à l'écran vidéo par l'intermédiaire de la routine video. Ces deux routines étant propres aux périphériques, elles ne seront pas décrites ici.

La codification en Morse aura la structure suivante sur un octet :

```
| 0   0   0   0   X   X   X   X |
+---+---+---+---+---+---+---+---+

          X = 0 ---> point
          X = 1 ---> trait
```

```
; transcodage d'un code numerique
; en code Morse.
;
DEB     CALL    CLAV     ; acquisition du caractere
                         ; au clavier. Resultat --> A
        LD      B,A      ; sauve caract dans B
        LD      HL,TABL  ; pointe debut table
SCRUT   LD      A,(HL)   ; extract caractere numerique
        OR      A        ; posit indicateurs
        JR      Z,FTAB   ; fin table atteinte (zero)
        CP      B        ; est-il egal à celui de B?
        INC     HL       ; à tout hasard, on prepare
        INC     HL       ; le caractere suivant
        JR      NZ,SCRUT ; non. on continue
        DEC     HL       ; c'est trouvé. Extraire
        LD      A,(HL)   ; le code Morse maintenant
        RET              ; et s'en retourner.
;
FTAB    LD      HL,MESER  ; pointe message erreur
        CALL    VIDEO    ; emission vers video
        JR      DEB      ; et continuer
;
TABL    DEFW    0F31H    ; 1  .----
        DEFW    0732H    ; 2  ..---
        DEFW    0333H    ; 3  ...--
        DEFW    0134H    ; 4  ....-
        DEFW    0035H    ; 5  .....
        DEFW    1036H    ; 6  -....
        DEFW    1837H    ; 7  --...
        DEFW    1C38H    ; 8  ---..
        DEFW    1E39H    ; 9  ----.
        DEFW    1F30H    ; 0  -----
        DEFB    0        ; fin table
;
MESER   DEFM    'ERREUR. CE CARACTERE N''EST PAS'
        DEFM    'DANS LA TABLE. RECOMMENCER.'
        DEFB    0        ; fin de message
```

Il ne faudra pas oublier qu'en mémoire, les éléments de la table seront rangés dans cet ordre :

```
TABL -----> 1 ASCII , code Morse du 1
TABL+2 ---> 2 ASCII , code Morse du 2
TABL+4 ---> 3 ASCII , code Morse du 3
            etc...
```

Avec ce principe de table, on peut transcoder n'importe quel caractère en n'importe quel autre. Pourquoi pas une conversion de clavier Anglo-saxon (QWERTY) en Européen (AZERTY)...

```
        CALL    CLAV            ; entrée clavier QWERTY
        LD      B,A             ; sauve caractere
        LD      HL,QWERTY       ; debut table QWERTY
        LD      DE,AZERTY       ; debut table AZERTY
SCRUT   LD      A,(HL)          ; extract caract
        OR      A               ; posit indic
        JR      Z,FTAB          ; si=0 fin table QWERTY
        CP      B               ; caractere trouvé?
        INC     HL              ; incremente pointeurs
        INC     DE              ; des 2 tables
        JR      NZ,SCRUT        ; si pas trouvé, suite.
        DEC     DE              ; stop. il est trouvé
        LD      A,(DE)          ; preleve le caractere
                                ; dans table AZERTY
        RET                     ; et l'on s'en va...
;
QWERTY  DEFM    'QWERTY....'    ; table QWERTY
        DEFB    0               ; fin table
AZERTY  DEFM    'AZERTY....'    ; table AZERTY
```

Ici, le principe est un peu différent : on utilise deux tables. Le pointeur dans la première table servira de pointeur dans la seconde.

Nous allons parler un peu des entrées/sorties maintenant, et plus particulièrement de ce que l'on appelle un ***driver***.

Le driver est un programme indépendant, qui a pour rôle de piloter un organe périphérique en déchargeant le programme utilisateur d'un certain nombre de tâches routinières.

Lorsque nous voulons, par exemple, envoyer un message sur une imprimante, nous fournissons l'adresse de ce message au driver correspondant. Ce qui se passe ensuite, nous ne voulons pas le savoir. Le principal, c'est que le message soit imprimé... !

Mais, qui s'occupera de savoir si l'imprimante est prête à recevoir chaque caractère du message ? Si elle arrive en bout de ligne, qui devra envoyer un ordre de positionnement à la ligne suivante, et en fin de page un positionnement à la page suivante ? C'est le driver imprimante.

Le programme utilisateur définit d'autre part, dans une zone spéciale appelée ***bloc de commande***, les paramètres utiles au driver, comme par exemple le nombre de lignes par page, le nombre de caractères par ligne, etc...

Pour imprimer un message, il suffira donc de faire :

```
LD        HL,MESSAG          ; adresse du message
          CALL     IMPRIM              ; appel driver
            ......
MESSAG    DEFM      'BAS LES PATTES...
```

Essayons d'imaginer ce que pourra être ce driver. La communication avec l'imprimante se fera par une instruction OUT à l'adresse 7 par exemple pour l'envoi du caractère, et l'instruction IN recevra l'état (ou status) de l'appareil sous la forme :

```
| 7   6   5   4   3   2   1   0 |
+---+---+---+---+---+---+---+---+
  ^       ^   ^
  !       !   !
  !       !   +- plus de papier si = 1
  !       +----- l'imprimante est en mode manuel si = 1
  !
  +------------- l'imprimante est occupée à imprimer si = 1
```

Nous définirons également un bloc de commande (CB) du driver imprimante ayant la structure suivante :

```
CB+0 ---> : nombre de lignes imprimables par page
CB+1 ---> : nombre de caracteres par lignes
CB+2 ---> : longueur de la marge basse
CB+3 ---> : compteur lignes/page (géré par driver)
CB+4 ---> : compteur caracts/ligne (géré par driver)
```

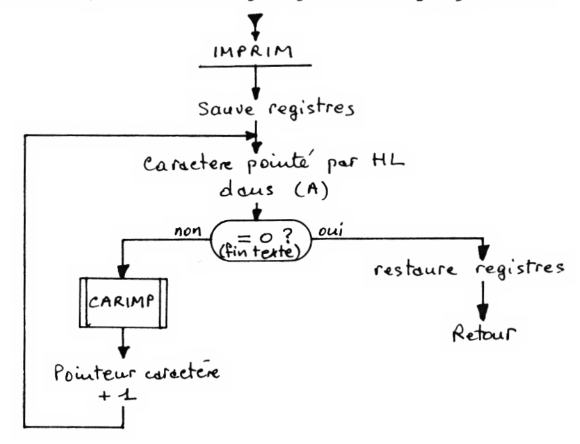
Avant d'appeler le driver, le programme utilisateur devra, évidemment, garnir les zones CB+0 à CB+2, afin de paramétrer son édition.

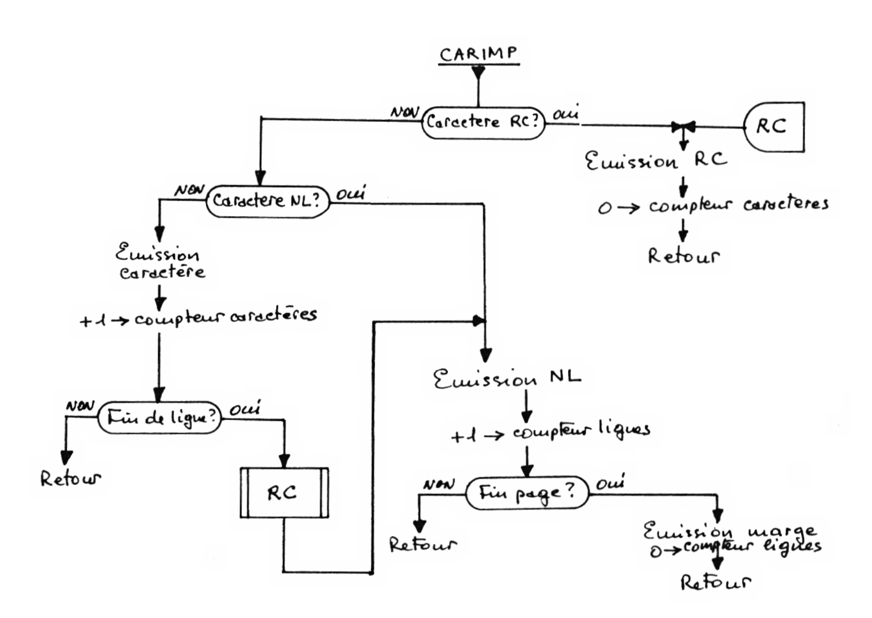
```
; EXEMPLE DE DRIVER IMPRIMANTE
; APPEL: (HL) = adresse du texte a imprimer
;          CALL   IMPRIM
;
IMPRIM  PUSH    AF          ; sauve les registres
        PUSH    BC          ; utilisés par le
        PUSH    HL          ; driver
        PUSH    IX          ;
        LD      IX,CB       ; adresse bloc commande
SUIT    LD      A,(HL)      ;
        OR      A           ; fin du texte?
        JR      NZ,IMP      ; non.
        POP     IX          ;
        POP     HL          ; restaure les registres
        POP     BC          ;
        POP     AF          ;
        RET                 ; et retour
IMP     CALL    CARIMP      ; impression caractere
        INC     HL          ; caractere suivant
        JR      SUIT        ;
;
; LOGIQUE D'IMPRESSION D'UN CARACTERE
;
CARIMP  CP      0DH         ; = retour chariot?
        JR      NZ,PACR     ; non
CR      LD      A,0DH       ; oui.
        CALL    EMIS        ; emission de RC
        LD      (IX+4),0    ; compteur caracteres= 0
        RET                 ;
PACR    CP      0AH         ; = nouvelle ligne?
        JR      Z,LF        ; oui
        CALL    EMIS        ; sinon emission caract.
        INC     (IX+4)      ; compteur carac + 1
        CP      (IX+1)      ; max atteint?
        RET     NZ          ; non
        CALL    CR          ; oui. passer à la ligne
        LD      A,0AH       ; code"nouvelle ligne"
LF      CALL    EMIS        ; emission NL
        INC     (IX+3)      ; compteur lignes+ 1
        CP      (IX+0)      ; page pleine?
        RET     NZ          ; non
        LD      B,(IX+2)    ; valeur marge
        LD      A,0AH       ; emission du
PAGE    CALL    EMIS        ; code NL
        DJNZ    PAGE        ; jusqu'à marge atteinte
        LD      (IX+3),0    ; compteur lignes = 0
        RET                 ;
;
EMIS    LD      C,A         ; sauve caractere
ATT     IN      A,(7)       ; lecture status imp.
        AND     0B0H        ; masque
        JR      NZ,ATT      ; non-disponible
        LD      A,C         ; caractere
        OUT     (7),A       ; emission imprimante
        RET                 ; et retour
;
; BLOC DE COMMANDE
;
CB      DEFB    60          ; lignes par page
        DEFB    50          ; caracteres par ligne
        DEFB    6           ; marge basse
        DEFS    2           ; compteurs
        END     CARIMP
```

Ce programme met abondamment en oeuvre la technique des sous-programmes : c'est tellement plus simple !

La lecture des états de l'imprimante est assez bestiale... et dans le cas où il n'y a plus de papier, il faudrait peut-être envoyer un message... sur l'écran, bien sûr ! A vous de l'améliorer !

Pour terminer, voici un organigramme du programme :





CHAPITRE 6

LE LOGICIEL LIÉ A L'ASSEMBLEUR

Il serait temps, alors que nous approchons la fin de cet ouvrage, de résumer les différentes opérations pouvant être effectuées sur un programme écrit en assembleur - et dans l'ordre chronologique, car nous avons parfois brûlé les étapes !

Avant d'assembler, donc, il faut créer le code source : c'est le but du programme nommé l'***éditeur de texte***. Il est ensuite possible de l'***assembler***.

Dans le cas où le code source est scindé en plusieurs parties distinctes, nous devons employer l'***éditeur de liens*** qui, partant des codes objet issus de l'assembleur, produira un code objet unique et exécutable.

Le programme peut alors être chargé en mémoire pour exécution. C'est le but du ***chargeur***.

Pour les pessimistes pris d'un doute affreux sur le travail effectué par les programmes ci-dessus, il existe un utilitaire nommé le ***désassembleur***. Ce programme peut aussi être utilisé par de petits curieux qui, ne possédant que le code objet d'un programme, voudraient bien savoir ce qu'il "a dans le ventre" !

Le désassembleur, en effet, effectue la fonction inverse de l'assembleur. Si ce dernier faisait la conversion : LD A,B en 78H, le désassembleur, partant de 78H, le convertira en LD A,B.

Evidemment, cette conversion n'est que partielle : il y manque les labels... et les commentaires !
Mais, commençons par le commencement.

L'EDITEUR DE TEXTE

Ce programme utilitaire peut parfois être inclus à l'assembleur (Editeur/Assembleur), mais on le rencontre plus fréquemment sous une forme autonome pouvant servir à d'autres fins (création de code source FORTRAN, BASIC et autres compilateurs).

Il permet de saisir un texte à partir d'un organe d'entrée (clavier en général) et de l'accumuler afin de former le code source. Les lignes d'entrée sont, en principe, numérotées automatiquement afin d'en faciliter le repérage.

Outre les fonctions de lecture/écriture sur support magnétique (ou autre), il offre un certain nombre de fonctions d'édition, destinées à faciliter la correction du programme source.

Il existe deux grands types d'éditeurs : l'***éditeur ligne*** et l'***éditeur page*** ou écran. Le premier est surtout utilisé en relation avec des organes d'édition sur papier (télétype par exemple), qui ne permettent pas d'effacer ou de corriger le texte introduit.

Le second s'adresse aux organes d'édition vidéo et permet l'effacement ou l'insertion d'un texte quelconque, à un endroit également quelconque, d'une page écran, en agissant simplement sur les déplacements curseur - chose impossible bien sûr avec l'éditeur du premier type.

Comme il y a une grande variété d'éditeurs de textes, allant des moins simples aux plus compliqués, nous nous contenterons d'énumérer les fonctions principales qu'ils peuvent réaliser :

- entrée d'un texte à partir de l'organe d'entrée,
- lecture d'un texte sur support magnétique,
- écriture d'un texte sur support magnétique,
- insertion ligne(s) ou caractère(s),
- remplacement ligne(s) ou caractère(s),
- recherche caractère(s),
- substitution de textes,
- positionnement sur numéro de ligne,
- positionnement sur caractère,
- effacement caractère(s) ou ligne(s),
- listage ligne(s),
- impression ligne(s) sur imprimante,
- renumérotation de ligne(s),
- macro-fonctions d'édition.

Mais si certains éditeurs sont souples, logiques et pratiques, il n'en est pas de même pour d'autres qui exigent beaucoup d'entraînement.

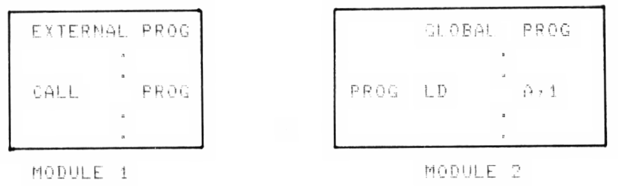
En règle générale, il vaut mieux éviter de faire des erreurs (!) pendant l'introduction du code source, pour éviter que les tentatives de correction n'apportent de nouvelles erreurs.

L'EDITEUR DE LIENS

Nous avons vu précédemment qu'un programme source, de par sa longueur ou pour des raisons pratiques, pouvait être décomposé en plusieurs modules, interconnectés par le truchement des directives EXTERNAL et GLOBAL.

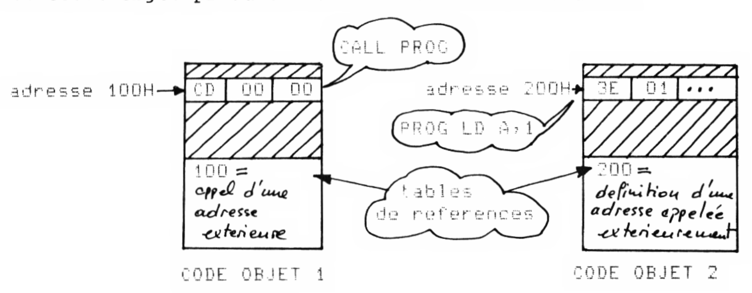
Ces modules sont alors assemblés séparément mais ne peuvent être exécutés, étant donné que certaines références externes ne sont pas satisfaites au niveau du code objet.

Par exemple :



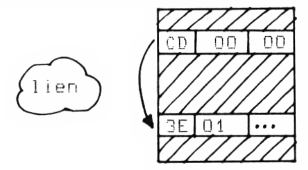
Dans cet exemple, le module 1 ne pourra être exécuté, puisque l'assembleur ne peut connaître la valeur réelle du label PROG, qui n'est défini que dans le module 2. Néanmoins, aucune erreur n'est signalée à l'assemblage, ce label étant défini comme externe.

Les codes objet produits auront alors la structure :



C'est à ce moment-là qu'intervient le grand Zorro...pardon : l'éditeur de liens, qui va satisfaire les références indéfinies entre les programmes objet.

L'unique module objet produit aura la structure suivante :



et sera, cette fois, exécutable.

Mais l'éditeur de liens nous offre une autre fonctionnalité intéressante. Lorsque l'on assemble un programme, il est possible de définir si le code objet produit doit être ***absolu*** ou ***translatable***.

Dans le premier cas, cela signifie que le programme ne pourra s'exécuter qu'aux adresses définies par rapport à la directive ORG, alors que dans le second cas, le code objet produit ne sera en aucun cas exécutable directement, mais devra être soumis à l'éditeur de liens, qui lui affectera une adresse d'origine définie à ce moment-là par l'utilisateur, et qui peut - en principe - être quelconque.

Bien entendu, le code hexadécimal figurant sur le listing d'assemblage, ainsi que les adresses correspondantes, ne seront plus à prendre en considération.

Dans le cas où plusieurs modules objet doivent fusionner, il suffit, soit que l'un d'eux soit absolu, soit de fixer une adresse d'origine au moment de l'édition de liens.

Les directives assembleur permettant de déterminer si un programme doit être absolu ou translatable sont :

```
ASEG    (ou equivalent) pour le code absolu
CSEG    (ou equivalent) pour le code translatable
```

et si nous n'en avons pas parlé jusqu'à maintenant, c'est que leur utilisation est spécifique à l'éditeur de liens.

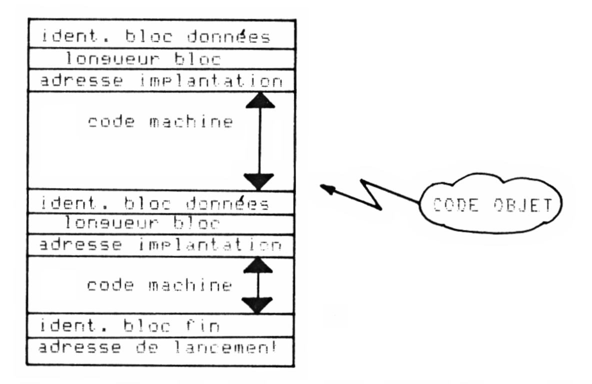
LE CHARGEUR

Lorsqu'un programme a été assemblé - et a éventuellement subi un traitement par l'éditeur de liens - il est prêt à être chargé en mémoire pour son exécution.

Ce chargement est réalisé par un programme nommé le ***chargeur*** qui exploite les informations contenues dans le code objet produit par l'assembleur ou l'éditeur de liens, qui lui permettront de placer correctement en mémoire le code exécutable du programme.

En général, le chargeur fait partie intégrante du système d'exploitation gérant l'ordinateur, et est activé par les ordres de type LOAD (chargement) ou RUN (chargement + lancement).

Nous allons examiner la structure générale du code objet d'un programme, afin de clarifier quelques points.



Cette structure peut varier d'un système à l'autre, mais le principe reste le même.

Nous voyons donc que le code objet est formé de blocs consécutifs, pouvant être (au moins) de deux types : bloc de données et bloc fin, qui sont différenciés par un octet d'identification, par exemple 01 pour le premier, 02 pour le second.

Le bloc de données contient deux informations concernant la longueur du bloc et son adresse d'implantation en mémoire.

Le bloc fin contient l'adresse à laquelle le programme devra être lancé au moment de son exécution.

En général, à chaque fois qu'une directive ORG est rencontrée dans le code source, l'assembleur crée un nouveau bloc. Dans le cas contraire, les blocs ont des longueurs standards de 128 ou 256 octets.

LE DESASSEMBLEUR

C'est l'outil de base des petits curieux ! L'assembleur à remonter le temps !

Comme nous l'avons déjà vu, cet utilitaire permet de convertir le code binaire en son code source d'origine...

Pour preuve, essayons de désassembler le binaire du programme ci-dessous ;

```
        ORG     1000H
DEB     LD      A,1       ; registre A = 1
        RET               ; retour
O.      DEFB    2         ; valeur = 2
        DEFM    'TEXTE'   ; c'est un texte!
        ....
        LD      A,(OCT)   ; (A) = octet
        ....
```

Nous obtiendrons :

```
1000H: 3E01     LD      A,1
1002H: C9       RET
1003H: 02       LD      (BC),A
1004H: 54       LD      D,H
1005H: 45       LD      B,L
1006H: 58       LD      E,B
1007H: 54       LD      D,H
1008H: 45       LD      B,L
  ...           ...     ...
XXXXH: 3A0301   LD      A,(1003H)
  ...           ...     ...
```

Il y a un fond de vérité, bien sûr, mais avouons-le, c'est assez troublant !

Il y a un truc très simple à savoir : lorsque le code paraît farfelu, et qu'il y a abondance d'instructions LD... essayez l'ASCII. C'est probablement une zone de données.

Ceci mis à part, on trouve de très bons désassembleurs, qui opèrent à partir d'un binaire chargé en mémoire ou à partir d'un code objet stocké sur disque.

Certains d'eux donnent même des noms de label du style : LAB1, LAB2, LAB3... et c'est toujours appréciable. D'autres (parfois les mêmes) dressent une table des références croisées (l'endroit où est défini un label et les endroits où on lui fait appel).

Il y en a même qui produisent un code source pouvant être directement ré-assemblé...! On n'arrête pas le progrès !

Le désassembleur, c'est le "passe-partout", la clé qui "ouvre" bien des programmes, et aucun n'y résiste, qu'il s'agisse de programmes réputés inviolables ou induplicables, ou qu'il s'agisse de programmes présentant des anomalies, comme cela arrive, hélas, trop souvent.

Combien de fois nous a t-il fallu recourir à ce moyen pour opérer de soi-disant "petites merveilles" et qui présentaient d'ignobles défauts.

Vous voulez apprendre à programmer en assembleur ? Apprenez aussi à désassembler !

CHAPITRE 7

CONNEXIONS AVEC LES LANGAGES EVOLUES

Les programmes écrits en assembleur sont généralement mis en oeuvre de deux manières différentes :

- le programme est autonome (exemple : utilitaire, compilateur, moniteur etc...) ;

- Il est dépendant d'un autre programme écrit en langage évolué (BASIC, FORTRAN, PASCAL...) et est souvent vu de ce dernier comme une simple routine, spécifique à une fonction précise.

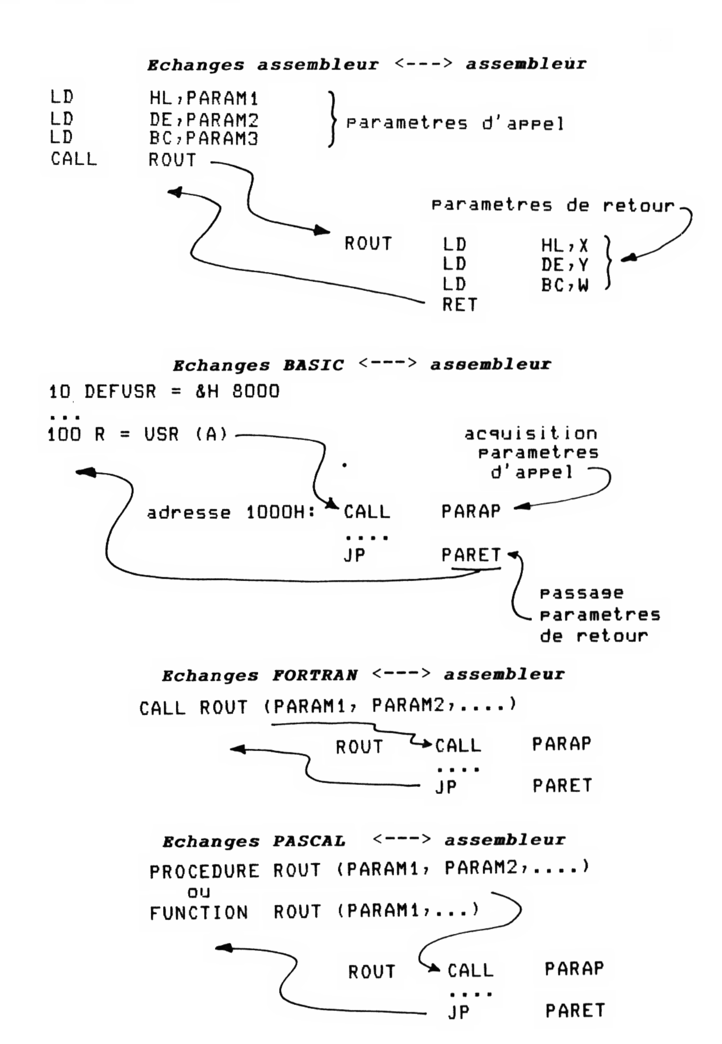
Un langage, tout évolué qu'il soit, ne peut réaliser toutes les fonctions, et en particulier celles fraîchement issues de l'imagination parfois délirante des programmeurs... !

Parfois également, telle ou telle fonction est conditionnée par le temps, et le langage évolué doit condescendre à faire appel au langage machine s'il veut mener cette tâche à bien.

Il y a aussi des méthodes plus terre-à-terre. Ainsi, le driver imprimante de votre BASIC n'est pas parfaitement adapté au modèle que vous possédez ? Il suffit de remplacer l'adresse du driver standard par celle du vôtre... et les commandes LPRINT seront traitées à votre convenance ! Cette opération s'effectue par la fonction Basic : POKE.

Vous vous souvenez de l'instruction assembleur CALL ? Il existe une fonction similaire dans presque tous les langages évolués, qui permet de faire appel à une routine écrite en assembleur, en lui passant un ou plusieurs paramètres. Après exécution de cette routine, un retour au langage évolué sera effectué, accompagné éventuellement d'un passage de paramètres.

Voyons quelques exemples :



Les labels PARAP et PARET représentent des points d'entrée de routines dont les adresses sont données par le concepteur du compilateur ou de l'interpréteur, et qui permettent le transfert des paramètres du langage évolué vers les registres du microprocesseur, et vice-versa.

Parfois, cette opération n'est pas nécessaire, et les paramètres sont directement placés dans les registres HL, DE et BC. Une autre variante est donnée par le registre HL qui

pointe une zone mémoire contenant une liste de paramètres. Ces modalités varient d'un système à l'autre, et ne sont données ici que pour l'exemple.

Le Fortran et le Pascal étant des langages compilés (les compilateurs sont des sortes d'assembleurs qui traduisent le code source évolué en code machine), il sera nécessaire d'utiliser les directives de références externes propres à ces langages, afin de créer des liens avec la routine assembleur. Une édition de liens suivra donc obligatoirement la phase de compilation.

En Basic, par contre (à moins d'avoir affaire à un compilateur) le problème sera différent. Le code objet de la routine devra être chargé séparément en mémoire et juxtaposé au code source du programme Basic interprété.

La commande Basic DEFUSR permettra de **DEF**inir l'adresse **U**tili**S**ateu**R** (jouant, ici, le rôle de l'éditeur de liens), et la fonction USR provoquera l'appel de cette routine. Le passage de paramètres se fera comme suit :

```
R = USR (A)
```

A représentant le paramètre à passer à la routine et R le paramètre de retour.

Lorsqu'un paramètre n'est pas suffisant, il est possible de passer l'adresse d'un tableau :

```
R = USR (VARPTR ( T (0) )
```

avec T(0) = paramètre 1 ; T(1) = paramètre 2 ... la routine se chargeant d'aller chercher les paramètres dans le tableau.

Mais prenons tout de suite un exemple très simple, qui consiste à multiplier un nombre par 2 au moyen d'une routine assembleur (programme réalisé sur TRS-80) :

```
10 DEFUSR = &H 8000
20 FOR AD = &H 8000 TO &H 8006
30 READ V : POKE AD,V : NEXT AD
40 INPUT "NOMBRE" ; N
50 PRINT N ; "* 2 =" ; USR (N)
60 GOTO 40
70 DATA 205 , 127 , 10 , 41 , 195 , 154 , 10
```

Routine reconstituée :

```
8000H: CD 7F 0A      CALL 0A7FH   ; passage param --> HL
8003H: 29            ADD  HL,HL   ; HL * 2
8004H: C3 9A 0A      JP   0A9AH   ; passage HL --> variable
```

Dans cet exemple, le code machine est chargé dynamiquement en mémoire par le programme Basic, au moyen de la boucle des lignes 20 et 30 qui garnit la zone 8000H à 8006H avec le code des instructions de la ligne 70 exprimées sous forme décimale.

Les appels 0A7FH et 0A9AH définis par les concepteurs du Basic TRS-80 permettent d'échanger les paramètres entre les deux programmes Basic et binaire.

Bien entendu, dans le cas où il n'y a pas de paramètre à passer, ces appels ne sont pas nécessaires, et il suffit que la routine se termine par l'instruction RET.

En partant d'un programme assemblé, la procédure sera :

- chargement du code objet de la routine (LOAD)
- chargement du programme BASIC ayant la structure :

```
10 DEFUSR = XXXX          (adresse de lancement routine)
   ....
100 A = USR ( B )          (appel de la routine)
```

Voilà ! Cet ouvrage est bientôt terminé. Son but initial a t-il été atteint : provoquer le déclic, l'étincelle (à ce sujet, ne le laissez jamais dans un endroit humide et à la portée des enfants) ?

Comme tout livre à vocation pédagogique, ce n'est pas un manuel de référence du Z80 (ce n'est pas une référence du tout, d'ailleurs) car il y manque un tas de choses.

Lorsque vous aurez lu ce livre, si votre curiosité est aiguisée et vous amène, tout naturellement, à aligner quelques instructions et à réfléchir dessus, c'est gagné !

Continuez ! Vous êtes sur la bonne voie ! Procurez-vous des listings de programmes, ou désassemblez les codes objet et essayez de comprendre leur fonctionnement et de les améliorer. Cela vaut un bon cours de perfectionnement. Alors la destinée de cet ouvrage, tout comme celle de l'étincelle, arrivera à son terme. Récupérez peut-être les annexes.

ANNEXE I

LES MATHEMATIQUES DE L'INFORMATIQUE

L'homme a dix doigts et depuis ce temps là, il compte habituellement en utilisant la ***base 10***.

Lorsque l'homme de Cromagnon, du haut de son rocher, voulait dire à son collègue se trouvant non loin de là, qu'il pouvait dénombrer 58 aurochs dans la plaine, il devait d'abord lever 8 doigts (unité), puis un instant plus tard, 5 doigts (dizaines, nombre de mains pleines) pour terminer par un geste démonstratif signifiant quelque chose comme "MIAM... MIAM !"

Il faut dire qu'ils avaient la vue perçante en ce temps là, mais tout de même, il y avait des limites, et lorsque la distance était trop importante, il fallait utiliser une autre méthode : les deux bras, par exemple.

Mais si, avec les deux mains, on pouvait compter jusqu'à 10 (base 10), avec les deux bras, on ne peut compter que jusqu'à 2 (base 2). De plus, il ne faut pas oublier que le nombre correspondant à la base n'est jamais utilisé : 10 aurochs auraient été codés avec les mains :

temps 1 : 0 doigt levé (unité)
temps 2 : 1 doigt levé (dizaine)
temps 3 : miam... miam !

De même, 58 aurochs seront codés comme suit :

```
temps 1: 8 unités                 --->   8
temps 2: 5 fois la base = 5 * 10 --->  50
temps 3: miam... miam!
```

Et s'il y en a 1253 :

```
temps 1: 3 unités                                        =     3
temps 2: 5 fois la base                = 5*10            =    50
temps 3: 2 fois la base fois la base   = 2*10*10         =   200
temps 4: 1 fois...                     = 1*10*10*10      =  1000
temps 5: miam...miam!
                                                          --------
                                                            1253
```

Avec les bras (base 2), 58 aurochs devront être codés :

```
temps:  1    2    3    4    5    6
```

C'est-à-dire :

```
temps 1: 0 unité                                          0
temps 2: 1 fois la base               = 1*2               2
temps 3: 0 fois la base fois la base  = 0*2*2             0
temps 4: 1 fois ....                  = 1*2*2*2           8
temps 5: 1 fois ....                  = 1*2*2*2*2        16
temps 6: 1 fois ....                  = 1*2*2*2*2*2      32
temps 7: miam... miam!                                ------
                                                         58
```

L'avantage de la méthode était évident : elle laissait à l'homme un bras libre, ce qui lui permettait d'en changer lorsqu'il y avait beaucoup d'aurochs, car dans ce cas là, la "transmission" était longue et fastidieuse...

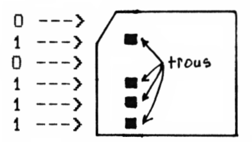
Ainsi, comme vous le voyez, 58 en base 10 et 111010 en base 2 sont deux représentations d'un même nombre (il y en a autant que de bases, c'est-à-dire un nombre infini). L'une est la représentation décimale, l'autre la représentation binaire.

Comme le bras de l'homme de Cro-moignon qui ne pouvait prendre que deux états (bras levé ou bras baissé), les informations traitées par l'ordinateur seront codées sur le même principe :

- **Etat 0** : pas de circulation de courant, pas de trou sur une carte ou un ruban perforé, pas de magnétisation sur une cassette ou une disquette.

- **Etat 1** : Circulation d'un courant, trou dans une carte ou un ruban perforé, magnétisation sur un support magnétique.

Le nombre 58 pourra alors être représenté en binaire sur une carte perforée :



La mémoire centrale est formée d'une mosaïque de cellules appelées ***bits***, chacun d'eux pouvant prendre l'un des deux états 0 ou 1 (bascule).

Habituellement, ces bits sont regroupés par huit pour former un ***octet***, et la plupart des microprocesseurs utilisent l'adressage octet, ce qui signifie que l'on accède simultanément à 8 bits de la mémoire, en lecture ou en écriture.

L'octet peut être représenté ainsi :

```
      |__|__|__|__|__|__|__|__|   (1 bit)
bit no: 7  6  5  4  3  2  1  0
```

et le nombre 58 pourra être codé :

```
 0  0  1  1  1  0  1  0
|__|__|__|__|__|__|__|__|
 7                    0
```

avec le bit 0 représentant le poids faible du nombre, et le bit 7 représentant le poids fort, la notion de poids dépendant essentiellement de la valeur de la puissance 2 du rang occupé par chaque bit :

bit 0 ---> 2^0 =1
bit 1 ---> 2^1 =2
bit 2 ---> 2^2 =4
bit 3 ---> 2^3 =8
bit 4 ---> 2^4 =16
bit 5 ---> 2^5 =32
bit 6 ---> 2^6 =64
bit 7 ---> 2^7 =128

Plus le rang est élevé, plus le poids est lourd. Notre nombre 58, dans sa représentation binaire, a les bits 1, 3, 4 et 5 positionnés à 1. Il est donc facile de retrouver sa valeur décimale :

```
bit 1 ---> 2
bit 3 ---> 8
bit 4 ---> 16
bit 5 ---> 32
          ----
           58
```

A l'inverse, un nombre décimal devra être divisé par les puissances successives de 2 (la base) afin de trouver sa configuration binaire.

Exemple : $133 = 2^7 + 2^2 + 2^0 = 128 + 4 + 1$

correspondant à :

```
        1  0  0  0  0  1  0  1
        L__L__L__L__L__L__L__L_J
bits: 7                 2     0
```

Mais cette représentation est lourde et encombrante pour l'homme. Il a donc été décidé de couper l'octet en 2 ***quartets*** (4 bits) :

```
 1  0  0  0    0  1  0  1
L__L__L__L_J  L__L__L__L_J
```

et de faire correspondre à chaque quartet 16 symboles différents et uniques :

```
0000 correspondra à :  0
0001                   1
0010                   2
0011                   3
0100                   4
0101                   5
0110                   6
0111                   7
1000                   8
1001                   9
1010 correspondra à : .....?
```

Les symboles numériques sont épuisés ? Qu'à cela ne tienne. Employons les alphabétiques :

```
1010 correspondra à :  A
1011                   B
1100                   C
1101                   D
1110                   E
1111                   F
```

Voilà ! Toutes les combinaisons possibles du quartet ont été écrites, et il faut s'arrêter là.

Par cette opération, nous venons, sans le savoir (vraiment ?) de passer de la base 2 (binaire) à la base 16 (hexadécimale), avec l'avantage d'une légère contraction d'écriture (15 devient F). Tous ces systèmes de numération ne sont, rappelons-le, que différents modes de représentation des mêmes nombres. Le tout est de connaître la base utilisée.

Un petit conseil : apprenez par coeur le tableau de correspondance binaire/hexadécimal. Allons... ce n'est pas si difficile !

Le nombre 58 (base 10) pourra être représenté par 3A en base 16, ce qui est tout de même plus agréable que 00111010... enfin, chacun ses goûts !

Bien entendu, au dela d'un octet, le processus continue :

```
|0  1  0  0|1  1  0  0|0  1  0  0|0  0  0  1|
    14        11 10        6               0
```

correspondra à 4C41 en base 16 (çà, c'est facile), et à :

$2^{14} + 2^{11} + 2^{10} + 2^{6} + 1$ en base 10, soit 19521.

Mais 19521... c'est 76 fois 256 plus 65,

et 76, c'est 4 fois 16 plus 12,

et 65, c'est 4 fois 16 plus 1

```
    4    C    4    1
```

Nous venons de repasser en base 16 ! Il est ensuite plus facile de basculer en base 2 si l'on connaît ses classiques :

```
  4    C    4    1
  ↓    ↓    ↓    ↓
0100 1100 0100 0001
```

Puisque nous en sommes au binaire, essayons d'aditionner deux nombres : 1 + 1 = 2... ?
Non. En base 2, le 2 n'existe pas. Il faut donc ajouter une tranche supplémentaire au résultat (comme pour 5 + 5 en base 10) :

```
   (1)
     1
 +   1
 -------
    1 0
```

Essayons encore :

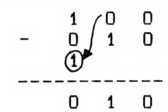
```
       (1)(1)(1)(1)
    1  0  1  1  0  1  0  0
 +  0  0  1  0  1  1  1  0
 -------------------------
    1  1  1  0  0  0  1  0
```

Le microprocesseur ne procède pas différemment pour effectuer une addition. En hexadécimal, c'est un peu moins simple, sauf si l'on sait compter sur ses doigts...

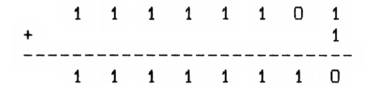
```
   (1)
    1 A 6 5
 +  0 C 3 5
 -----------
    2 6 9 A
```

On compte comme en base 10, mais au dela de 9, on continue par A. Il n'y a une retenue qu'au dela de F.

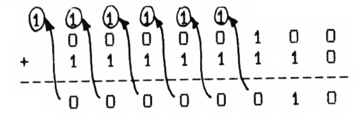
Et la soustraction binaire ?



Bien sûr ! Mais il est plus facile (tout au moins pour le microprocesseur) d'utiliser ce que l'on nomme le ***complément à 2***. Le complément à 2, c'est le complément à 1 plus 1 !
Le complément à 1 consiste, lui, à inverser (complémenter) la valeur de chaque bit du nombre sur une longueur égale au format dans lequel il est représenté (habituellement 8 ou 16). Ainsi, 010 doit être représenté 00000010 si l'on travaille sur des nombres de 8 bits. Son complément à 1 sera : 11111101, et son complément à 2 :



Effectuons maintenant une addition de ce complément avec la valeur 100 de l'exemple précédent :



et cela nous donne le même résultat que la soustraction 0100 - 0010. Voila comment une soustraction se transforme en addition.

Il faut tout de même noter que la dernière retenue "tombe dans le vide" mais ce n'est pas inquiétant puisque le résultat est aussi sur un format de 8 bits.

Cela nous conduit maintenant à aborder la notion de ***nombre signé***. En informatique, il y a deux façons de considérer un nombre : en valeur absolue (tous les bits sont significatifs) et en valeur signée. Dans ce dernier cas, le poids fort du nombre indiquera si celui-ci est négatif (sous forme complément à 2 si = 1) ou positif (poids fort = 0).

Par exemple, le nombre binaire :

1 1 1 1 1 1 1 0

(FE en base 16), sera égal à 254 en valeur décimale absolue, et à -2 en valeur décimale signée, car il représente, dans ce cas là, le complément à 2 du nombre 2 (notez que l'addition d'un nombre et de son complément à 2 donne toujours zéro). C'est facile : Il suffit de s'expliquer au départ sur la valeur de la base et sur le type de représentation...

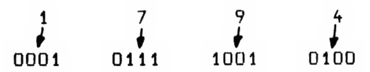
Un autre type de représentation est quelquefois utilisé en informatique : c'est la représentation ***BCD***, qui signifie ***Décimal Code Binaire***

On l'emploie en général dans des programmes exécutant des opérations directement en décimal, sans passer par le binaire.

Les chiffres décimaux prennent les valeurs 0 à 9, on fait correspondre, à chaque chiffre du nombre, un quartet codé ainsi :

```
0000 ---> 0
0001 ---> 1
0010 ---> 2
 ......
1001 ---> 9
```

La codification est la même que pour l'hexadécimal, mais cette fois on s'arrête à 9. Ainsi, le nombre 1794 sera représenté :



Pour terminer, nous allons dire un mot au sujet des ***opérateurs logiques***.
Il existe 4 fonctions logiques fréquemment utilisées sur les nombres binaires : le ***NON*** (***NOT***), le ***ET*** (***AND***), le ***OU*** (***OR***) et le ***OU EXCLUSIF*** (***XOR***).

Le ***NON***, nous l'avons déjà vu : c'est le complément à 1.
Le ***ET*** (ou intersection logique) de deux nombres binaires s'effectue selon la règle suivante :

```
0 et 0 ---> 0
0 et 1 ---> 0
1 et 0 ---> 0
1 et 1 ---> 1
```

appliquée à chaque couple de bits des deux nombres à intersecter.

Exemple :

```
     1  0  1  0  1  1  0  1
ET   0  1  1  1  1  0  1  1
    ------------------------
     0  0  1  0  1  0  0  1
```

Sous forme hexadécimale, nous écrirons que : AD AND 7B = 29.
Quoi de plus simple ?

Le bit résultant sera à 1 si le bit du premier nombre ET le bit du second nombre sont à 1.

Le ***OU*** (ou réunion logique) obéit à la règle :

```
0 et 0 ---> 0
0 et 1 ---> 1
1 et 0 ---> 1
1 et 1 ---> 1
```

Exemple :

```
     1  0  1  0  1  1  0  1
OU   1  1  0  0  0  1  0  1
    ------------------------
     1  1  1  0  1  1  0  1
```

que l'on peut aussi écrire : AD OR C5 = ED en hexadécimal. Le bit résultant sera à un si le bit du premier nombre OU le bit du second nombre est à 1.

Le ***OU exclusif*** (ou disjonction logique) obéit à la même règle que le précédent, sauf pour la dernière condition :

0 et 0 ---> 0
0 et 1 ---> 1
1 et 0 ---> 1
1 et 1 ---> 0

Pour que le bit résultant soit à 1, il faut donc que le bit du premier nombre OU le bit du second nombre soit à 1, mais pas les deux en même temps.

Exemple :

```
         0 1 0 1 1 1 0 1
OU EX    0 1 1 0 1 0 0 0
        -----------------
         0 0 1 1 0 1 0 1
```

ANNEXE II

CORRIGES DES EXERCICES

CHAPITRE II

EXERCICE 2.1 :

```
SOM     INC     B           ; +1 dans B
        DEC     C           ; -1 dans C
        JR      NZ,SOM      ; Jusqu'a ce que C = 0
```

EXERCICE 2.2 :

Lors de l'appel de la routine par le CALL ROUT, l'adresse de l'instruction d'arrêt JRØ est mise en pile, et un branchement est effectué à l'adresse ROUT. Là, l'adresse de retour est dépilée (POP HL), incrémentée de 2 et remise en pile (PUSH). Lors du RET (retour), le programme se poursuivra donc deux octets plus loin que ce qui était prévu, c'est-à-dire à l'adresse ROUT. Le JRØ n'est pas exécuté. Le POP HL "récupère" OABCH qui est incrémenté de 2, et le retour (RET) provoquera une poursuite du programme à l'adresse OABEH.

EXERCICE 2.3 :

Ce programme vient tout simplement se modifier lui-même. Le code 3DH correspondant à l'instruction DEC A (voir liste en annexe) est écrit à la place de l'instruction INC A située à l'adresse ADR. Le registre A sera donc décrémenté puis incrémenté, et retrouvera donc sa valeur initiale de 3DH !

EXERCICE 2.4 :

Ce ne sont pas les solutions qui manquent, et vous en avez certainement trouvé une !

Voici un programme qui serait séduisant... s'il marchait parfaitement : il utilise 2 instructions... qui dit mieux ? Il remet à zéro la totalité de la mémoire moins... 4 octets, et commence alors à se détruire lui-même ! A ce moment là, cela ne devient pas triste du tout... Bien que simple, son mécanisme est assez complexe. Essayez tout de même de le comprendre jusqu'au bout, et plus particulièrement à partir du moment ou le CALL ADR devient un CALL 00...
Pour fonctionner (enfin presque), ce programme doit être implanté à l'adresse 0FFFAH, de façon que l'adresse de retour du CALL soit zéro !

```
FFFA  31FCFF           LD      SP,ADR-1
FFFD  CDFDFF    ADR    CALL    ADR
0000            ; fin de la memoire. On repart a 0
```

Amusant, non ?

EXERCICE 2.5 :

```
        LD      C,5             ; compteur
        LD      HL,BUFF         ; pointe debut buffer
        LD      DE,BUFF+9       ; pointe fin buffer
ECHANG  LD      A,(DE)          ; caracteres à
        LD      B,(HL)          ; permuter
        LD      (HL),A          ; (DE) ---> (HL)
        LD      A,B             ;
        LD      (DE),A          ; (HL) ---> (DE)
        INC     HL              ; debut + 1
        DEC     DE              ; fin - 1
        DEC     C               ; 5 fois
        JR      NZ,ECHANG       ; suite si non zero
                                ; fin
```

CHAPITRE III

EXERCICE 3.1 :

```
; conversion Hexa Ascii ---> Binaire
;
HEXBIN  SUB     30H     ;
        CP      10      ; est-ce un chiffre?
        RET     C       ; oui. Retour.
        SUB     7       ; c'est une lettre (A à F)
        RET             ;
```

EXERCICE 3.2 :

```
        LD      A,B     ;
        CALL    HEXBIN  ; conversion poids forts
        RLCA            ; decaler dans
        RLCA            ; le quartet
        RLCA            ; fort
        RLCA            ; de A
        LD      B,A     ; A ---> B
        LD      A,C     ; poids faibles
        CALL    HEXBIN  ; conversion
        OR      B       ; reunion des deux
                        ; fin
```

EXERCICE 3.3 :

```
; en sortie, Z = 1 si egalité
;            C = 1 si DE > HL
;
                LD      A,H
                SUB     D
                RET     NZ
                LD      A,L
                SUB     E
                RET
```

EXERCICE 3.4 :

```
; en sortie, Z = 1 si egalité
;
        COMPAR  LD      A,(DE)
                CP      (HL)
                RET     NZ
                DEC     C
                RET     Z
                INC     HL
                INC     DE
                JR      COMPAR
```

Mais on peut aussi utiliser l'instruction CPI :

```
COMPAR  LD      B,0
        LD      A,(DE)
        CPI
        RET     NZ
        RET     PO
        INC     DE
        JR      COMPAR
```

EXERCICE 3.5 :

```
MUL10   LD      D,H
        LD      E,L       ; HL ---> DE
        ADD     HL,HL     ; HL * 2
        ADD     HL,HL     ; HL * 4
        ADD     HL,DE     ; HL * 5
        ADD     HL,HL     ; HL * 10
```

A N N E X E III

LE CODE ASCII

POIDS forts / faibles		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SP	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENG	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	↑	n	~
F	1111	SI	VS	/	?	O	←	o	DEL

Exemple d'utilisation:

A = 41H
? = 3FH

Signification des fonctions usuelles:

BEL: cloche, avertisseur sonore
BS : backspace, recul curseur
HT : tabulation horizontale
LF : line feed, retour ligne
VT : tabulation verticale
CR : carriage return, retour chariot (retour ligne)
SP : space, barre d'espace

ANNEXE IV

LE JEU D'INSTRUCTIONS DU Z80

Pour la petite histoire, voici comment cette liste a été créée.

a. Un programme BASIC, partant de la racine de chaque mnémonique, va produire la famille correspondante.

Exemple : LD <SR>,<SR>

(SR représente les simples registres), va créer tous les codes : LD A,A, LD A,B ... LD L,L.

Le fichier ainsi constitué est placé sur disque.

b. Tri du fichier par ordre croissant.

c. Numérotation des instructions (programme BASIC).

d. Assemblage du fichier pour générer le code machine (listing sur fichier disque).

e. Suppressions des numéros de lignes (devenus inutiles) par programme BASIC.

f. Altération du code machine (dd, nn, etc...) par un logiciel de traitement de texte.

g. Disposition du texte en deux colonnes de 60 lignes par page (programme BASIC).

h. Edition de la liste (voir ci-après).

Les symboles employés ont la signification suivante :

dd : valeur de déplacement signée sur 8 bits,

N : valeur de 8 bits de l'opérande,

nn : valeur de 8 bits dans le code machine,

ADR : label d'adresse en opérande,

aaaa: valeur d'adresse sur 16 bits dans le code machine (poids faibles + poids forts),

DEPL: déplacement dans l'opérande des instructions en mode relatif,

NN : valeur de 16 bits ou adresse dans l'opérande,

nnnn: valeur de 16 bits dans le code machine (poids faibles + poids forts).

```
0000 8E        ADC A,(HL)      *  0066 DDCBdd4E  BIT 1,(IX+dd)
0001 DD8Edd    ADC A,(IX+dd)   *  006A FDCBdd4E  BIT 1,(IY+dd)
0004 FD8Edd    ADC A,(IY+dd)   *  006E CB4F      BIT 1,A
0007 8F        ADC A,A         *  0070 CB48      BIT 1,B
0008 88        ADC A,B         *  0072 CB49      BIT 1,C
0009 89        ADC A,C         *  0074 CB4A      BIT 1,D
000A 8A        ADC A,D         *  0076 CB4B      BIT 1,E
000B 8B        ADC A,E         *  0078 CB4C      BIT 1,H
000C 8C        ADC A,H         *  007A CB4D      BIT 1,L
000D 8D        ADC A,L         *  007C CB56      BIT 2,(HL)
000E CEnn      ADC A,N         *  007E DDCBdd56  BIT 2,(IX+dd)
0010 ED4A      ADC HL,BC       *  0082 FDCBdd56  BIT 2,(IY+dd)
0012 ED5A      ADC HL,DE       *  0086 CB57      BIT 2,A
0014 ED6A      ADC HL,HL       *  0088 CB50      BIT 2,B
0016 ED7A      ADC HL,SP       *  008A CB51      BIT 2,C
0018 86        ADD A,(HL)      *  008C CB52      BIT 2,D
0019 DD86dd    ADD A,(IX+dd)   *  008E CB53      BIT 2,E
001C FD86dd    ADD A,(IY+dd)   *  0090 CB54      BIT 2,H
001F 87        ADD A,A         *  0092 CB55      BIT 2,L
0020 80        ADD A,B         *  0094 CB5E      BIT 3,(HL)
0021 81        ADD A,C         *  0096 DDCBdd5E  BIT 3,(IX+dd)
0022 82        ADD A,D         *  009A FDCBdd5E  BIT 3,(IY+dd)
0023 83        ADD A,E         *  009E CB5F      BIT 3,A
0024 84        ADD A,H         *  00A0 CB58      BIT 3,B
0025 85        ADD A,L         *  00A2 CB59      BIT 3,C
0026 C6nn      ADD A,N         *  00A4 CB5A      BIT 3,D
0028 09        ADD HL,BC       *  00A6 CB5B      BIT 3,E
0029 19        ADD HL,DE       *  00A8 CB5C      BIT 3,H
002A 29        ADD HL,HL       *  00AA CB5D      BIT 3,L
002B 39        ADD HL,SP       *  00AC CB66      BIT 4,(HL)
002C DD09      ADD IX,BC       *  00AE DDCBdd66  BIT 4,(IX+dd)
002E DD19      ADD IX,DE       *  00B2 FDCBdd66  BIT 4,(IY+dd)
0030 DD29      ADD IX,IX       *  00B6 CB67      BIT 4,A
0032 DD39      ADD IX,SP       *  00B8 CB60      BIT 4,B
0034 FD09      ADD IY,BC       *  00BA CB61      BIT 4,C
0036 FD19      ADD IY,DE       *  00BC CB62      BIT 4,D
0038 FD29      ADD IY,IY       *  00BE CB63      BIT 4,E
003A FD39      ADD IY,SP       *  00C0 CB64      BIT 4,H
003C A6        AND (HL)        *  00C2 CB65      BIT 4,L
003D DDA6dd    AND (IX+dd)     *  00C4 CB6E      BIT 5,(HL)
0040 FDA6dd    AND (IY+dd)     *  00C6 DDCBdd6E  BIT 5,(IX+dd)
0043 A7        AND A           *  00CA FDCBdd6E  BIT 5,(IY+dd)
0044 A0        AND B           *  00CE CB6F      BIT 5,A
0045 A1        AND C           *  00D0 CB68      BIT 5,B
0046 A2        AND D           *  00D2 CB69      BIT 5,C
0047 A3        AND E           *  00D4 CB6A      BIT 5,D
0048 A4        AND H           *  00D6 CB6B      BIT 5,E
0049 A5        AND L           *  00D8 CB6C      BIT 5,H
004A E6nn      AND N           *  00DA CB6D      BIT 5,L
004C CB46      BIT 0,(HL)      *  00DC CB76      BIT 6,(HL)
004E DDCBdd46  BIT 0,(IX+dd)   *  00DE DDCBdd76  BIT 6,(IX+dd)
0052 FDCBdd46  BIT 0,(IY+dd)   *  00E2 FDCBdd76  BIT 6,(IY+dd)
0056 CB47      BIT 0,A         *  00E6 CB77      BIT 6,A
0058 CB40      BIT 0,B         *  00E8 CB70      BIT 6,B
005A CB41      BIT 0,C         *  00EA CB71      BIT 6,C
005C CB42      BIT 0,D         *  00EC CB72      BIT 6,D
005E CB43      BIT 0,E         *  00EE CB73      BIT 6,E
0060 CB44      BIT 0,H         *  00F0 CB74      BIT 6,H
0062 CB45      BIT 0,L         *  00F2 CB75      BIT 6,L
0064 CB4E      BIT 1,(HL)      *  00F4 CB7E      BIT 7,(HL)
```

```
00F6 DDCBdd7E   BIT 7,(IX+dd)  *  0163 D9         EXX
00FA FDCBdd7E   BIT 7,(IY+dd)  *  0164 76         HALT
00FE CB7F       BIT 7,A        *  0165 ED46       IM 0
0100 CB78       BIT 7,B        *  0167 ED56       IM 1
0102 CB79       BIT 7,C        *  0169 ED5E       IM 2
0104 CB7A       BIT 7,D        *  016B ED78       IN A,(C)
0106 CB7B       BIT 7,E        *  016D DBnn       IN A,(N)
0108 CB7C       BIT 7,H        *  016F ED40       IN B,(C)
010A CB7D       BIT 7,L        *  0171 ED48       IN C,(C)
010C DCaaaa     CALL C,ADR     *  0173 ED50       IN D,(C)
010F FCaaaa     CALL M,ADR     *  0175 ED58       IN E,(C)
0112 D4aaaa     CALL NC,ADR    *  0177 ED60       IN H,(C)
0115 CDaaaa     CALL ADR       *  0179 ED68       IN L,(C)
0118 C4aaaa     CALL NZ,ADR    *  017B 34         INC (HL)
011B F4aaaa     CALL P,ADR     *  017C DD34dd     INC (IX+dd)
011E ECaaaa     CALL PE,ADR    *  017F FD34dd     INC (IY+dd)
0121 E4aaaa     CALL PO,ADR    *  0182 3C         INC A
0124 CCaaaa     CALL Z,ADR     *  0183 04         INC B
0127 3F         CCF            *  0184 03         INC BC
0128 BE         CP (HL)        *  0185 0C         INC C
0129 DDBEdd     CP (IX+dd)     *  0186 14         INC D
012C FDBEdd     CP (IY+dd)     *  0187 13         INC DE
012F BF         CP A           *  0188 1C         INC E
0130 B8         CP B           *  0189 24         INC H
0131 B9         CP C           *  018A 23         INC HL
0132 BA         CP D           *  018B DD23       INC IX
0133 BB         CP E           *  018D FD23       INC IY
0134 BC         CP H           *  018F 2C         INC L
0135 BD         CP L           *  0190 33         INC SP
0136 FEnn       CP N           *  0191 EDAA       IND
0138 EDA9       CPD            *  0193 EDBA       INDR
013A EDB9       CPDR           *  0195 EDA2       INI
013C EDA1       CPI            *  0197 EDB2       INIR
013E EDB1       CPIR           *  0199 E9         JP (HL)
0140 2F         CPL            *  019A DDE9       JP (IX)
0141 27         DAA            *  019C FDE9       JP (IY)
0142 35         DEC (HL)       *  019E DAaaaa     JP C,ADR
0143 DD35dd     DEC (IX+dd)    *  01A1 FAaaaa     JP M,ADR
0146 FD35dd     DEC (IY+dd)    *  01A4 D2aaaa     JP NC,ADR
0149 3D         DEC A          *  01A7 C3aaaa     JP ADR
014A 05         DEC B          *  01AA C2aaaa     JP NZ,ADR
014B 0B         DEC BC         *  01AD F2aaaa     JP P,ADR
014C 0D         DEC C          *  01B0 EAaaaa     JP PE,ADR
014D 15         DEC D          *  01B3 E2aaaa     JP PO,ADR
014E 1B         DEC DE         *  01B6 CAaaaa     JP Z,ADR
014F 1D         DEC E          *  01B9 38dd       JR C,DEPL
0150 25         DEC H          *  01BB 18dd       JR DEPL
0151 2B         DEC HL         *  01BD 30dd       JR NC,DEPL
0152 DD2B       DEC IX         *  01BF 20dd       JR NZ,DEPL
0154 FD2B       DEC IY         *  01C1 28dd       JR Z,DEPL
0156 2D         DEC L          *  01C3 02         LD (BC),A
0157 3B         DEC SP         *  01C4 12         LD (DE),A
0158 F3         DI             *  01C5 77         LD (HL),A
0159 10dd       DJNZ DEPL      *  01C6 70         LD (HL),B
015B FB         EI             *  01C7 71         LD (HL),C
015C E3         EX (SP),HL     *  01C8 72         LD (HL),D
015D DDE3       EX (SP),IX     *  01C9 73         LD (HL),E
015F FDE3       EX (SP),IY     *  01CA 74         LD (HL),H
0161 08         EX AF,AF'      *  01CB 75         LD (HL),L
0162 EB         EX DE,HL       *  01CC 36nn       LD (HL),N
```

```
01CE DD77dd     LD (IX+dd),A    *  0256 4C         LD C,H
01D1 DD70dd     LD (IX+dd),B    *  0257 4D         LD C,L
01D4 DD71dd     LD (IX+dd),C    *  0258 0Enn       LD C,N
01D7 DD72dd     LD (IX+dd),D    *  025A 56         LD D,(HL)
01DA DD73dd     LD (IX+dd),E    *  025B DD56dd     LD D,(IX+dd)
01DD DD74dd     LD (IX+dd),H    *  025E FD56dd     LD D,(IY+dd)
01E0 DD75dd     LD (IX+dd),L    *  0261 57         LD D,A
01E3 DD36ddnn   LD (IX+dd),N    *  0262 50         LD D,B
01E7 FD77dd     LD (IY+dd),A    *  0263 51         LD D,C
01EA FD70dd     LD (IY+dd),B    *  0264 52         LD D,D
01ED FD71dd     LD (IY+dd),C    *  0265 53         LD D,E
01F0 FD72dd     LD (IY+dd),D    *  0266 54         LD D,H
01F3 FD73dd     LD (IY+dd),E    *  0267 55         LD D,L
01F6 FD74dd     LD (IY+dd),H    *  0268 16nn       LD D,N
01F9 FD75dd     LD (IY+dd),L    *  026A ED5Bnnnn   LD DE,(NN)
01FC FD36ddnn   LD (IY+dd),N    *  026E 11nnnn     LD DE,NN
0200 32nnnn     LD (NN),A       *  0271 5E         LD E,(HL)
0203 ED43nnnn   LD (NN),BC      *  0272 DD5Edd     LD E,(IX+dd)
0207 ED53nnnn   LD (NN),DE      *  0275 FD5Edd     LD E,(IY+dd)
020B 22nnnn     LD (NN),HL      *  0278 5F         LD E,A
020E DD22nnnn   LD (NN),IX      *  0279 58         LD E,B
0212 FD22nnnn   LD (NN),IY      *  027A 59         LD E,C
0216 ED73nnnn   LD (NN),SP      *  027B 5A         LD E,D
021A 0A         LD A,(BC)       *  027C 5B         LD E,E
021B 1A         LD A,(DE)       *  027D 5C         LD E,H
021C 7E         LD A,(HL)       *  027E 5D         LD E,L
021D DD7Edd     LD A,(IX+dd)    *  027F 1Enn       LD E,N
0220 FD7Edd     LD A,(IY+dd)    *  0281 66         LD H,(HL)
0223 3Annnn     LD A,(NN)       *  0282 DD66dd     LD H,(IX+dd)
0226 7F         LD A,A          *  0285 FD66dd     LD H,(IY+dd)
0227 78         LD A,B          *  0288 67         LD H,A
0228 79         LD A,C          *  0289 60         LD H,B
0229 7A         LD A,D          *  028A 61         LD H,C
022A 7B         LD A,E          *  028B 62         LD H,D
022B 7C         LD A,H          *  028C 63         LD H,E
022C ED57       LD A,I          *  028D 64         LD H,H
022E 7D         LD A,L          *  028E 65         LD H,L
022F 3Enn       LD A,N          *  028F 26nn       LD H,N
0231 ED5F       LD A,R          *  0291 2Annnn     LD HL,(NN)
0233 46         LD B,(HL)       *  0294 21nnnn     LD HL,NN
0234 DD46dd     LD B,(IX+dd)    *  0297 ED47       LD I,A
0237 FD46dd     LD B,(IY+dd)    *  0299 DD2Annnn   LD IX,(NN)
023A 47         LD B,A          *  029D DD21nnnn   LD IX,NN
023B 40         LD B,B          *  02A1 FD2Annnn   LD IY,(NN)
023C 41         LD B,C          *  02A5 FD21nnnn   LD IY,NN
023D 42         LD B,D          *  02A9 6E         LD L,(HL)
023E 43         LD B,E          *  02AA DD6Edd     LD L,(IX+dd)
023F 44         LD B,H          *  02AD FD6Edd     LD L,(IY+dd)
0240 45         LD B,L          *  02B0 6F         LD L,A
0241 06nn       LD B,N          *  02B1 68         LD L,B
0243 ED4Bnnnn   LD BC,(NN)      *  02B2 69         LD L,C
0247 01nnnn     LD BC,NN        *  02B3 6A         LD L,D
024A 4E         LD C,(HL)       *  02B4 6B         LD L,E
024B DD4Edd     LD C,(IX+dd)    *  02B5 6C         LD L,H
024E FD4Edd     LD C,(IY+dd)    *  02B6 6D         LD L,L
0251 4F         LD C,A          *  02B7 2Enn       LD L,N
0252 48         LD C,B          *  02B9 ED4F       LD R,A
0253 49         LD C,C          *  02BB ED7Bnnnn   LD SP,(NN)
0254 4A         LD C,D          *  02BF F9         LD SP,HL
0255 4B         LD C,E          *  02C0 DDF9       LD SP,IX
```

```
02C2 FDF9      LD SP,IY       *  0334 CB8B      RES 1,E
02C4 31nnnn    LD SP,NN       *  0336 CB8C      RES 1,H
02C7 EDA8      LDD            *  0338 CB8D      RES 1,L
02C9 EDB8      LDDR           *  033A CB96      RES 2,(HL)
02CB EDA0      LDI            *  033C DDCBdd96  RES 2,(IX+dd)
02CD EDB0      LDIR           *  0340 FDCBdd96  RES 2,(IY+dd)
02CF ED44      NEG            *  0344 CB97      RES 2,A
02D1 00        NOP            *  0346 CB90      RES 2,B
02D2 B6        OR (HL)        *  0348 CB91      RES 2,C
02D3 DDB6dd    OR (IX+dd)     *  034A CB92      RES 2,D
02D6 FDB6dd    OR (IY+dd)     *  034C CB93      RES 2,E
02D9 B7        OR A           *  034E CB94      RES 2,H
02DA B0        OR B           *  0350 CB95      RES 2,L
02DB B1        OR C           *  0352 CB9E      RES 3,(HL)
02DC B2        OR D           *  0354 DDCBdd9E  RES 3,(IX+dd)
02DD B3        OR E           *  0358 FDCBdd9E  RES 3,(IY+dd)
02DE B4        OR H           *  035C CB9F      RES 3,A
02DF B5        OR L           *  035E CB98      RES 3,B
02E0 F6nn      OR N           *  0360 CB99      RES 3,C
02E2 EDBB      OTDR           *  0362 CB9A      RES 3,D
02E4 EDB3      OTIR           *  0364 CB9B      RES 3,E
02E6 ED79      OUT (C),A      *  0366 CB9C      RES 3,H
02E8 ED41      OUT (C),B      *  0368 CB9D      RES 3,L
02EA ED49      OUT (C),C      *  036A CBA6      RES 4,(HL)
02EC ED51      OUT (C),D      *  036C DDCBddA6  RES 4,(IX+dd)
02EE ED59      OUT (C),E      *  0370 FDCBddA6  RES 4,(IY+dd)
02F0 ED61      OUT (C),H      *  0374 CBA7      RES 4,A
02F2 ED69      OUT (C),L      *  0376 CBA0      RES 4,B
02F4 D3nn      OUT (N),A      *  0378 CBA1      RES 4,C
02F6 EDAB      OUTD           *  037A CBA2      RES 4,D
02F8 EDA3      OUTI           *  037C CBA3      RES 4,E
02FA F1        POP AF         *  037E CBA4      RES 4,H
02FB C1        POP BC         *  0380 CBA5      RES 4,L
02FC D1        POP DE         *  0382 CBAE      RES 5,(HL)
02FD E1        POP HL         *  0384 DDCBddAE  RES 5,(IX+dd)
02FE DDE1      POP IX         *  0388 FDCBddAE  RES 5,(IY+dd)
0300 FDE1      POP IY         *  038C CBAF      RES 5,A
0302 F5        PUSH AF        *  038E CBA8      RES 5,B
0303 C5        PUSH BC        *  0390 CBA9      RES 5,C
0304 D5        PUSH DE        *  0392 CBAA      RES 5,D
0305 E5        PUSH HL        *  0394 CBAB      RES 5,E
0306 DDE5      PUSH IX        *  0396 CBAC      RES 5,H
0308 FDE5      PUSH IY        *  0398 CBAD      RES 5,L
030A CB86      RES 0,(HL)     *  039A CBB6      RES 6,(HL)
030C DDCBdd86  RES 0,(IX+dd)  *  039C DDCBddB6  RES 6,(IX+dd)
0310 FDCBdd86  RES 0,(IY+dd)  *  03A0 FDCBddB6  RES 6,(IY+dd)
0314 CB87      RES 0,A        *  03A4 CBB7      RES 6,A
0316 CB80      RES 0,B        *  03A6 CBB0      RES 6,B
0318 CB81      RES 0,C        *  03A8 CBB1      RES 6,C
031A CB82      RES 0,D        *  03AA CBB2      RES 6,D
031C CB83      RES 0,E        *  03AC CBB3      RES 6,E
031E CB84      RES 0,H        *  03AE CBB4      RES 6,H
0320 CB85      RES 0,L        *  03B0 CBB5      RES 6,L
0322 CB8E      RES 1,(HL)     *  03B2 CBBE      RES 7,(HL)
0324 DDCBdd8E  RES 1,(IX+dd)  *  03B4 DDCBddBE  RES 7,(IX+dd)
0328 FDCBdd8E  RES 1,(IY+dd)  *  03B8 FDCBddBE  RES 7,(IY+dd)
032C CB8F      RES 1,A        *  03BC CBBF      RES 7,A
032E CB88      RES 1,B        *  03BE CBB8      RES 7,B
0330 CB89      RES 1,C        *  03C0 CBB9      RES 7,C
0332 CB8A      RES 1,D        *  03C2 CBBA      RES 7,D
```

```
03C4 CBBB        RES 7,E       *  043F C7         RST 00
03C6 CBBC        RES 7,H       *  0440 CF         RST 08
03C8 CBBD        RES 7,L       *  0441 D7         RST 10H
03CA C9          RET           *  0442 DF         RST 18H
03CB D8          RET C         *  0443 E7         RST 20H
03CC F8          RET M         *  0444 EF         RST 28H
03CD D0          RET NC        *  0445 F7         RST 30H
03CE C0          RET NZ        *  0446 FF         RST 38H
03CF F0          RET P         *  0447 9E         SBC A,(HL)
03D0 E8          RET PE        *  0448 DD9Edd     SBC A,(IX+dd)
03D1 E0          RET PO        *  044B FD9Edd     SBC A,(IY+dd)
03D2 C8          RET Z         *  044E 9F         SBC A,A
03D3 ED4D        RETI          *  044F 98         SBC A,B
03D5 ED45        RETN          *  0450 99         SBC A,C
03D7 CB16        RL (HL)       *  0451 9A         SBC A,D
03D9 DDCBdd16    RL (IX+dd)    *  0452 9B         SBC A,E
03DD FDCBdd16    RL (IY+dd)    *  0453 9C         SBC A,H
03E1 CB17        RL A          *  0454 9D         SBC A,L
03E3 CB10        RL B          *  0455 DEnn       SBC A,N
03E5 CB11        RL C          *  0457 ED42       SBC HL,BC
03E7 CB12        RL D          *  0459 ED52       SBC HL,DE
03E9 CB13        RL E          *  045B ED62       SBC HL,HL
03EB CB14        RL H          *  045D ED72       SBC HL,SP
03ED CB15        RL L          *  045F 37         SCF
03EF 17          RLA           *  0460 CBC6       SET 0,(HL)
03F0 CB06        RLC (HL)      *  0462 DDCBddC6   SET 0,(IX+dd)
03F2 DDCBdd06    RLC (IX+dd)   *  0466 FDCBddC6   SET 0,(IY+dd)
03F6 FDCBdd06    RLC (IY+dd)   *  046A CBC7       SET 0,A
03FA CB07        RLC A         *  046C CBC0       SET 0,B
03FC CB00        RLC B         *  046E CBC1       SET 0,C
03FE CB01        RLC C         *  0470 CBC2       SET 0,D
0400 CB02        RLC D         *  0472 CBC3       SET 0,E
0402 CB03        RLC E         *  0474 CBC4       SET 0,H
0404 CB04        RLC H         *  0476 CBC5       SET 0,L
0406 CB05        RLC L         *  0478 CBCE       SET 1,(HL)
0408 07          RLCA          *  047A DDCBddCE   SET 1,(IX+dd)
0409 ED6F        RLD           *  047E FDCBddCE   SET 1,(IY+dd)
040B CB1E        RR (HL)       *  0482 CBCF       SET 1,A
040D DDCBdd1E    RR (IX+dd)    *  0484 CBC8       SET 1,B
0411 FDCBdd1E    RR (IY+dd)    *  0486 CBC9       SET 1,C
0415 CB1F        RR A          *  0488 CBCA       SET 1,D
0417 CB18        RR B          *  048A CBCB       SET 1,E
0419 CB19        RR C          *  048C CBCC       SET 1,H
041B CB1A        RR D          *  048E CBCD       SET 1,L
041D CB1B        RR E          *  0490 CBD6       SET 2,(HL)
041F CB1C        RR H          *  0492 DDCBddD6   SET 2,(IX+dd)
0421 CB1D        RR L          *  0496 FDCBddD6   SET 2,(IY+dd)
0423 1F          RRA           *  049A CBD7       SET 2,A
0424 CB0E        RRC (HL)      *  049C CBD0       SET 2,B
0426 DDCBdd0E    RRC (IX+dd)   *  049E CBD1       SET 2,C
042A FDCBdd0E    RRC (IY+dd)   *  04A0 CBD2       SET 2,D
042E CB0F        RRC A         *  04A2 CBD3       SET 2,E
0430 CB08        RRC B         *  04A4 CBD4       SET 2,H
0432 CB09        RRC C         *  04A6 CBD5       SET 2,L
0434 CB0A        RRC D         *  04A8 CBDE       SET 3,(HL)
0436 CB0B        RRC E         *  04AA DDCBddDE   SET 3,(IX+dd)
0438 CB0C        RRC H         *  04AE FDCBddDE   SET 3,(IY+dd)
043A CB0D        RRC L         *  04B2 CBDF       SET 3,A
043C 0F          RRCA          *  04B4 CBD8       SET 3,B
043D ED67        RRD           *  04B6 CBD9       SET 3,C
```

```
04B8 CBDA        SET 3,D         *  052C CB20        SLA B
04BA CBDB        SET 3,E         *  052E CB21        SLA C
04BC CBDC        SET 3,H         *  0530 CB22        SLA D
04BE CBDD        SET 3,L         *  0532 CB23        SLA E
04C0 CBE6        SET 4,(HL)      *  0534 CB24        SLA H
04C2 DDCBddE6    SET 4,(IX+dd)   *  0536 CB25        SLA L
04C6 FDCBddE6    SET 4,(IY+dd)   *  0538 CB2E        SRA (HL)
04CA CBE7        SET 4,A         *  053A DDCBdd2E    SRA (IX+dd)
04CC CBE0        SET 4,B         *  053E FDCBdd2E    SRA (IY+dd)
04CE CBE1        SET 4,C         *  0542 CB2F        SRA A
04D0 CBE2        SET 4,D         *  0544 CB28        SRA B
04D2 CBE3        SET 4,E         *  0546 CB29        SRA C
04D4 CBE4        SET 4,H         *  0548 CB2A        SRA D
04D6 CBE5        SET 4,L         *  054A CB2B        SRA E
04D8 CBEE        SET 5,(HL)      *  054C CB2C        SRA H
04DA DDCBddEE    SET 5,(IX+dd)   *  054E CB2D        SRA L
04DE FDCBddEE    SET 5,(IY+dd)   *  0550 CB3E        SRL (HL)
04E2 CBEF        SET 5,A         *  0552 DDCBdd3E    SRL (IX+dd)
04E4 CBE8        SET 5,B         *  0556 FDCBdd3E    SRL (IY+dd)
04E6 CBE9        SET 5,C         *  055A CB3F        SRL A
04E8 CBEA        SET 5,D         *  055C CB38        SRL B
04EA CBEB        SET 5,E         *  055E CB39        SRL C
04EC CBEC        SET 5,H         *  0560 CB3A        SRL D
04EE CBED        SET 5,L         *  0562 CB3B        SRL E
04F0 CBF6        SET 6,(HL)      *  0564 CB3C        SRL H
04F2 DDCBddF6    SET 6,(IX+dd)   *  0566 CB3D        SRL L
04F6 FDCBddF6    SET 6,(IY+dd)   *  0568 96          SUB (HL)
04FA CBF7        SET 6,A         *  0569 DD96dd      SUB (IX+dd)
04FC CBF0        SET 6,B         *  056C FD96dd      SUB (IY+dd)
04FE CBF1        SET 6,C         *  056F 97          SUB A
0500 CBF2        SET 6,D         *  0570 90          SUB B
0502 CBF3        SET 6,E         *  0571 91          SUB C
0504 CBF4        SET 6,H         *  0572 92          SUB D
0506 CBF5        SET 6,L         *  0573 93          SUB E
0508 CBFE        SET 7,(HL)      *  0574 94          SUB H
050A DDCBddFE    SET 7,(IX+dd)   *  0575 95          SUB L
050E FDCBddFE    SET 7,(IY+dd)   *  0576 D6nn        SUB N
0512 CBFF        SET 7,A         *  0578 AE          XOR (HL)
0514 CBF8        SET 7,B         *  0579 DDAEdd      XOR (IX+dd)
0516 CBF9        SET 7,C         *  057C FDAEdd      XOR (IY+dd)
0518 CBFA        SET 7,D         *  057F AF          XOR A
051A CBFB        SET 7,E         *  0580 A8          XOR B
051C CBFC        SET 7,H         *  0581 A9          XOR C
051E CBFD        SET 7,L         *  0582 AA          XOR D
0520 CB26        SLA (HL)        *  0583 AB          XOR E
0522 DDCBdd26    SLA (IX+dd)     *  0584 AC          XOR H
0526 FDCBdd26    SLA (IY+dd)     *  0585 AD          XOR L
052A CB27        SLA A           *  0586 EEnn        XOR N
```

Achevé d'imprimer en décembre 1981
sur les presses de l'imprimerie Laballery et Cie
58500 Clamecy
Dépôt légal : 4e trimestre 1981

N° d'impression : 20310
N° d'édition : 86470-14-3
ISBN : 2. 86595.014.X



Programmer en Assembleur

illustré avec le jeu d'instruction du Z-80

Cet ouvrage, qui s'adresse aux lecteurs connaissant déjà un langage tel Basic, constitue une introduction complète au langage machine, et à son frère l'assembleur, comprenant des exercices et des exemples. Bien qu'illustré par le code du Z 80, il sera d'une lecture tout aussi utile aux possesseurs de P.S.I. disposant d'un autre microprocesseur.

Editions du P.S.I.
B.P. 86
77400 Lagny/Marne

ISBN 2-86470-014-X

26-9907
Imprimé en France


Alain Pinaud
P.S.I.
Programmer en Assembleur



Document numérisé avec amour par

AMSTRAD CPC

MÉMOIRE ÉCRITE



https://acpc.me/